AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Electronics, Communications and Automation
Department of Automation and Systems Technology

Miguel Pérez Cardoso

Development of an upper-level software of a ceiling-mounted home prototype robot.

Espoo January 16, 2011

Supervisors:

Professor Aarne Halme
Aalto University

Professor Elisa Maria Ruiz Navas
University Carlos III of Madrid

Instructor:

David Leal Martinez
Aalto University

Universidad
Carlos III de Madrid

Acknowledgements

It is impossible to think in this Thesis without Klaus Müller. His work, as part of the Software Team, was fundamental, and thanks of him, today I am writing this. I also want to be grateful with the rest of the Ceilbot project, especially with the Professor Halme, for his advices at the beginning and the end of the project, and to give me the opportunity to be part of this project, with Tomi, for all his work during the semester, coordinating all the teams, and with Antti Maula for his enormous patient, giving to me all his knowledge of MaCI and GIMnet. Special mention has my instructor, David. He has been working with me in every step of the project, from almost the first day I arrived in Finland, until the end, focusing this Thesis on a concrete road.

My friends are also part of this Thesis, helping me from years to get this point. I would like to named of my friends from the school and the university, but it is impossible. However, I would like to stress my grateful to Tony, Eva, Shoji, Pablis, and specially, Antonio. More than a friend is Millán.

No one has worked more than my parents for seeing this moment. From them I have learnt that with effort, honesty and humility, everything is possible. They are the example of my life. Due to my brother, Mario, every day is more difficult than the previous one. Nothing is more complicated than trying to be an example for him.

More than two years ago my life change forever. I meet Carolina, my girlfriend. She has suffered and fought for seeing this day with me. Thank you very much.

Espoo, January 16, 2011

Miguel Pérez Cardoso

Aalto University: School of Science and Technology

Abstract of the Master's Thesis

Author:	Miguel Pérez Cardoso		
Title of the thesis:	Development of an upper-level software of a ceiling-mounted home prototype robot		
Date:	January 16, 2011	**Number of pages:**	56
Faculty:	Faculty of Electronics, Communications and Automation		
Department:	Automation and System Technology		
Professorship:	Automation Technology (Aut-84)		
Supervisors:	Professor Aarne Halme (TKK) & Elisa María Ruiz Navas(UC3M)		
Instructor:	David Leal Martínez		

The objective of this thesis is to design, implement and test an upper level software of a robot mounted on a rail system. This upper-level software has three main purposes:

- To determinate the fastest route between the actual position of the robot and the target point. This task is reached using the Dijkstra's Algorithm.
- To determinate the most efficient target point according on the shape and weight of the object to manipulate.
- To establish the communication between the different programs using MaCI.

This thesis is integrated in the work of the Ceilbot's Software Group. Since the Ceilbot robot is not yet built, the software, was tested using the Field Service Robots Simulator.

Keywords:	Rail system, Dijsktra, Ceilbot, Closest Path, MaCI, GimNet

Contents

List of Figures

Chapter 1

Introduction

This Thesis is framed within the Ceilbot Group, a project aiming to develop a robot working on the ceiling, which at the beginning may not seem usual. However, the ceiling provides several features which make it an excellent environment for service robots.

- The ceiling is almost static and clean, whereas the floor is chaotic. With no humans or pets around it, the ceiling has no changes, and for that reason the possibilities of motion freedom are higher.
- There is high energy availability and, the ceiling allows, for different and innovative ideas, to supply it.

The Ceilbot project started in Fall 2009 with the design of different Ceilbots on different environments. This Thesis is done in Fall 2010 and at this moment the objectives of the entire group were to design, implement and test the first prototype of a Ceilbot mounted on a rail system for a home environment. To accomplish this goal the work was divided among several teams and this Thesis is part of the work developed by the Software Team.

The objectives of the Software Team were:

1. To setup a communication infrastructure using GIMnet.

2. To design and implement a basic extendable graphical user interface (GUI).

3. To design and implement an upper-level software (ULS).

The first task was achieved by the entire Software Team. GIMnet and MaCI are powerful and innovative tools, created by Aalto University. GIMnet is a communication infrastructure designed mainly for service robots, and MaCI is a complete library of interfaces between GIMnet and different hardwares.

This Thesis focuses on the development of the ULS, and although the process was separete from the development of GUI, continuous references to the GUI will be done. ULS is in charge of the path planning and behaviour of the robot. This ULS has three different objectives:

- To determinate the fastest route between the actual position of the robot and the target point. This task is reached using the Dijkstra's Algorithm.
- To determinate the most efficient target point according on the shape and weight of the object to manipulate.
- To establish the communication between the different programs using MaCI.

During the semester different parts of the robot were developed but the first prototype was not finished. To try to avoid this problem the work was done using one simulator, the Field Service Robots Simulator. The main problem with this simulator is that it has no map (such as a rail system) and, it was necessary to simulate it with different maps which provide some of the specific features of the Ceilbot map doing with them all the necessary tests. Thanks to them, the Ceilbot environment was simulated and the objectives were reached.

For a correct understanding of the Thesis it is necessary to go one step back and explain the beginning of the Ceilbot project and the work done by the rest of the teams during Fall 2010, as this Thesis would not be complete without the whole information of the Ceilbot.

1.1 Background

The Ceilbot project started in September 2009 with a brainstorming of a group of students trying to develop the main applications of the robot in different environments, the desired features, the communication and different issues about safety. The work was done by nine students who were organized in five different teams, developing a different environment or function each group. These teams were:

- Home robot team
- Sports hall robot team
- Fire fighter robot team
- Mapping team
- House construction consultation

For several months each team designed all the different features of each Ceilbot, developing the basic model, functionalities, and services of the robot, providing to the future Ceilbot's teams the first approach of the project.

This Master's Thesis is the continuation of the work of one of these teams, the Home Robot Team. However, the work of all the teams is going to be explained on this document, because all they developed interesting features and ideas for the Ceilbot, even, which though might have been developed for other environments, were taken in consideration for this Thesis. The entire information of the Ceilbot Project is available in the project website (Ceilbot, 2010).

1.1.1 Home Robot Team

The home robot team developed a concept to be utilized at home for fetching things, cleaning, and vacuuming, as possible tasks. It utilizes a sophisticated trunk-like actuator. This Thesis is integrated under this design.

Possible tasks

All the possible tasks are divided in two sections, services and cleaning. The services would be done directely by the manipulator, whereas the cleaning tasks would require to add additional hardware to the manipulator. Also the movement of the robot along the room is the first and the most basic task. It needs mapping and recognizing to reach all these tasks.

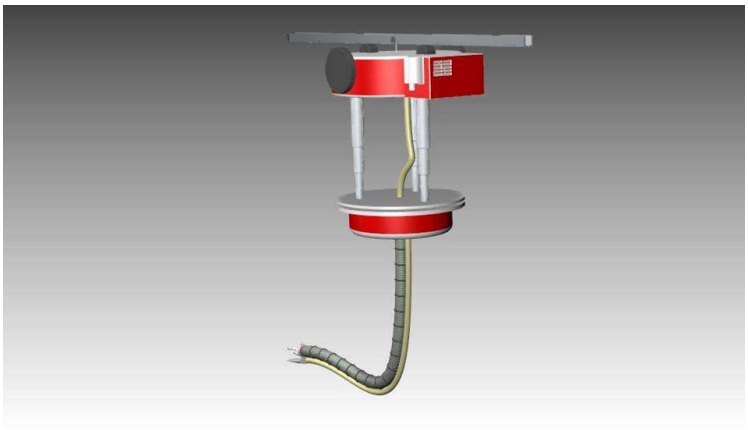


Figure 1.1: Digital model of the first Home Ceilbot concept

The main service tasks of the robot is the motion freedom around the room and to have the possibility of bringing objects, feeding animals or watering flowers.

For the cleaning tasks, some examples could be vacuuming, wiping dust or washing windows. As we can see, for all of these tasks a specific module is needed, so the final design must take this into consideration.

Rail System

The robot is going to move along a rail system. The rail system would have straight parts, corners, and cross-sections. It should be designed taking care of the robot's need to access every point of the whole room.

This Ceilbot is going to be in a home environment, so it needs to move along different rooms. For this, it is necessary to design a system for crossing the

doorways. The team decided that the best solution is using doorways able to turn, allowing to cross them.

Trunk

The solution is for the manipulator to be in the shape of an elephant's trunk. This trunk would be flexible and could easily reach every point or object in a room. The basic idea is presented in figure 1.2:

Figure 1.2: Example of a possible model of manipulator based on an Elephant's trunk

1.2 Related background

The work of the rest of the groups is also an important part of the project and for that reason a small summary of all of them are explained in this section.

1.2.1 Sports Halls Robot

This Ceilbot was thinked for working on a Sport Hall or Stadium, able to help in different sports environments. The group decided to plan all this work in four different steps.

- Definition and Environment
- Qualitative Concept
- Modelisation
- Deeper Design

Definition and Environment

One of the main characteristics of every Sport Hall is the big size of the room. In order to design how to fix the robot on the ceiling, the team decided to use a cable system. This system provide to the robot a large motion freedom allowing it, to reach every point of the Sport Hall or Stadium. The idea was use cables attached to all sides of the hall and an actuator to move the robot.

Qualitative Concept

After defining the environment, the next step was to choose a task for Ceilbot in the Sports Hall. The flexibility of the robot and its ability to adapt to the situation make it a powerful tool which could record in real time, help the coach to analyze the game, clean the floor or the ceiling, or gather balls in a match. Moreover, there are different features in the robot that must be considered, such as the installation or the stability. These issues must be covered in all future work.

Modelation

The figure 1.3 is an example about how should the robot be. It is also an example about other functionalities, such as being a tennis opponent and following the ball with a video camera.

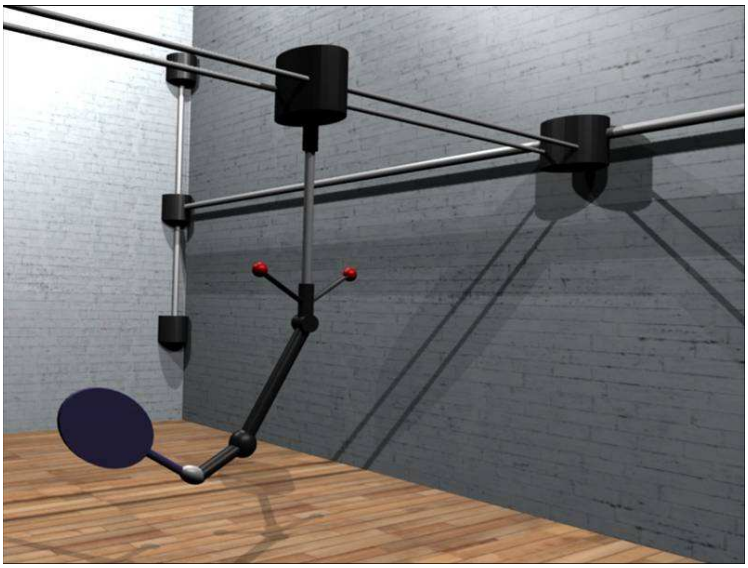


Figure 1.3: Digital model of the first Sports Hall Ceilbot concept

Deeper Design

One of the top challengues of the Sports Halls Ceilbot is the stabilization. It is vital to have one system for that. In this case the group decided to use an electronic stabilizator. It uses four motors and four accelerometers with which they calculate the inertia and balance of the robot.

For controlling the motor driver, the robot would have four winches. One main computer would control the robot communicatying with it by a CAN bus. Through this bus the robot power supply would also done.

1.2.2 Fire Fighter Robot

This Ceilbot has two very specific tasks: monitoring the environment, and detecting fire. To accomplish this the robot is going to use different commercial smoke and heat detectors placed along the room, and the robot also will have an automatic firefighting arm to extinguish the flames.

Challenges

According to its specific features, the main challenges for the Fire Fighter Ceilbot are:

- To be able to move with an automatic system of navigation, avoiding every element which could be on the ceiling, or different eventualities which could appear in the room.
- To have an automatic arm with a tool for extinguishing the flame.
- To have an entire sensing system to determine its own and the flame's position.
- To have a remote control.

Motion possibilities

The group decided to attach the robot with beams. Even when it is necessary to build a beam system this option is cheap and simple, and it has two advantages: it allows total navigation, and it is only necessary to use energy for motion, not for holding its own weight. Small engines would supply the motion.

Robot structure

The robot would have one arm which would provide the "up and down" motion, and using the sensors with a gradient algorithm, it could estimate the source of the flame.

The robot would use batteries for power supply, so, the robot could recharge itself in a corner of the room (placed designed for it). The digital model of this Ceilbot is shown in figure 1.4.

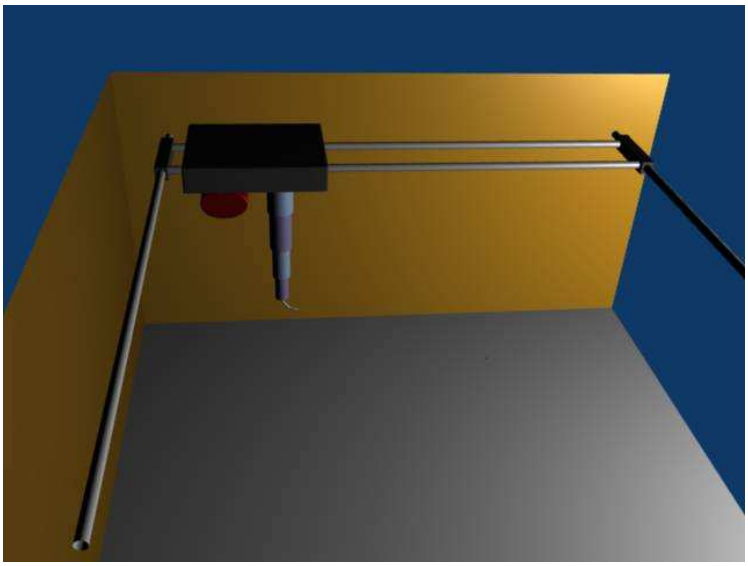


Figure 1.4: Digital model of the first Fire Fighter Ceilbot concept

User interface

When the robot detects fire it would notify the fire brigade. This communication would be via GSM so they would have all the information in real time. Moreover, the user should have the option of moving the robot.

1.2.3 Mapping Robot

The mapping team designed a ceil-walking robot, and concentrated on sensing and navigation activities. This group, instead of working on other environments, focused on the location and mapping system and also studied different methods of locomotion and attachment of the robot on the ceiling. All this work could be implemented in any different environment.

The attachment and locomotion system

Two different challenges must be attended: the attachment of the robot to the ceiling, even in a power failure, and the motion freedom. To overcome the first challenge, the team decided to focus on two different versions:

The first one, uses four chains and two actuators. The movement along the room is provided by moving the actuators, and for moving the robot on the other direction it is necessary to control the length of the chains. The main limitation of this system is that it is only possible to use it in one room, due to the impossibility of passing the chain system through doorways.

The other option is to design a robot with four arms which would be able to grip to different metal anchor pins along the ceiling. The robot could move around the room climbing from pin to pin. This system provides perfect motion freedom and could also use the walls if necessary. This second version is shown in figure 1.5.

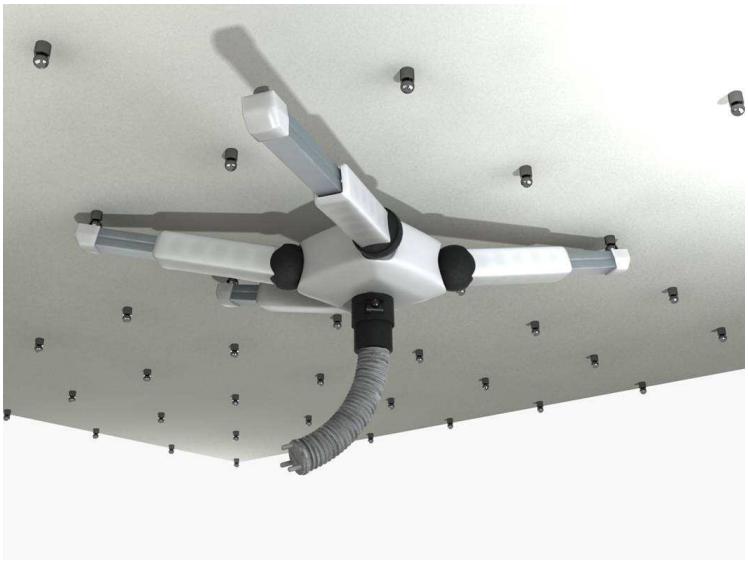


Figure 1.5: Digital model of the first Arms Design Ceilbot concept

The mapping and locating system

The main idea of the mapping algorithm is to create a three-layer environment deppending on the different frequency of change of all the elements of the environment. The structure would have one high level, one medium level, and one low level map. The first one would contain permanent elements such as doorways or walls; the second one would contain elements which should not change but are not fixed, like furniture; and finally, the low level map would contain the most recently measured data. The idea of this structure is to do the navigation as fast as possible.

The sensor system

The system would be formed by proximity sensors, a multi-axis accelerometer and gyroscope, position information from the robot's servos, a video camera, and a three-dimensional spatial scanner. These last two, could work together if the scanning was performed using structured light triangulation.

Safety of operation

In this subject it is basic to provide a list of specific characteristics which Ceilbot must have for creating an environment where it do not disturb any human activity. The sensor's system would provide useful information about the environment that the robot must interpret in order to achieve this task. Moreover, the robot must have an emergency stop, and also voice commands could be used for emergency stopping.

The pin system and the chain system, must be strong enough to carry the robot with any other element that it would need to keep, or move.

1.3 First Prototype

After the brainstorming, the work continued with the creation of the first Ceilbot's working prototype. The Home environment was selected and the aim of the project was to design , build, implement, and test the robot during the academic year 2010 - 2011.

The construction of the robot was organized among five different teams, each with at least one student. All these teams worked together developing the robot, using as background the work of the previous semesters. According with this, the robot would have an elephant's trunk as a manipulator, and for the motion along the room, there would be one trolley mounted on a rail system. The teams and the task of each of them were:

- Infrastructure Team: To design and build a modular structure of the rail system mounting.
- Rail System Team: To design and build the rail system.
- Localization Team: To design the localization and mapping system.
- Software Team: To design and implement the Graphical User Interface and the Upper Level Software.
- Trunk Team: To design and build one manipulator based on the elephant's trunk.

The next sections are going to be a summary of the work done during the Fall 2010, with the only exception of the Software Team. This Thesis is part of the work of this team, so it will be explained extensively along the next chapters. For more information of any of the different teams, in the project website (Ceilbot, 2010), in the section Fall 2010, is possible to check the entire work.

1.3.1 Infrastructure

The design of the infrastructure started in the Spring 2010, and continued in Fall 2010. The final design is one grid of three by three meters, formed by elements of one meter of length. The grid is made of aluminum, with a profile of 40 by 40 millimeters. The profile characteristics are: (in figure 1.6 is shown the profile).

- Material: Aluminum, anodized
- Height: 40 mm
- Width: 40 mm
- Cross-sectional area: 9.16 cm2
- Moment of Inertia, x-axis: 13.96 cm4

- Moment of Inertia, y-axis: 13.96 cm4
- Moment of Inertia, torsional: 1.93 cm4
- Resistance Moment, x-axis: 6.98 cm3
- Resistance Moment, y-axis: 6.98 cm3
- Weight, spec. Length: 2.47 kg/m

Figure 1.6: Profile used for the construction of the ceiling. Beam Profile 40x40

The grid was calculated to be enough strong to support the rail system and the Ceilbot. This grid was assembled and fixed on the ceiling. The assembled grid on the floor of the working room is shown in figure 1.7.

Figure 1.7: Grid Assembled

For the attachment of the grid to the ceiling, it was necessary to use U-shaped components and some screws.

1.3.2 Rail System

The final rail system would have two different phases. The difference between the first one and the second one ,is that the last one has a turntable which permits the selection of different routes and gives a higher level of freedom to the robot.

At the beginning the project would used the first phase of the track routing. After testing the Ceilbot and demostrate that everything is working correctly, the rail system would be the route phase two. Both designs permit reach every point of the two rooms, and also permit to work with the doorway and the implementation of it, to the final Ceilbot, so it has all the possible elements which would be in future real applications. Both designs are shown in figure 1.8.

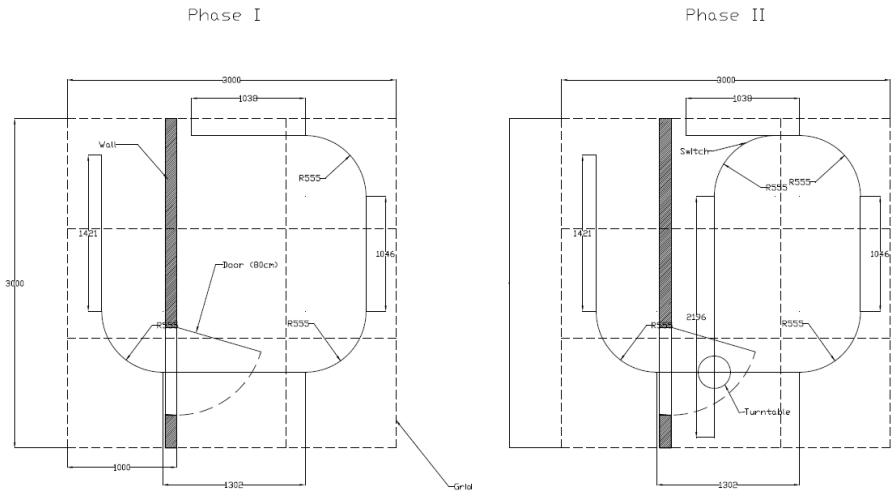


Figure 1.8: The two phases of the track routing

1.3.3 Localization

This group worked on two different topics at the same time: mapping and odometry, and both parts should work together. The first one is the continuation of the work already started in Spring 2010, while the odometry part started in the Fall 2010.

Odometry

Under the asumption that the robot would not be faster than 2 m/s, the odometry should help to avoid the following problems:

- Sensor error: There could be some noise in the measurement.
- Slippage: When the robot turns a corner, it could slips, so the measurement would be incorrect.
- Error in the estimation of the wheel diameter.

The solution that the team decided, was to use two incremental encoders, one in the motor axes, and the other one in an opposite wheel. This solution is inexpensive, and after different tests done by the team, the measurements were more accurate.

Mapping

During the Spring of 2010 this team worked on the mapping algorithm, whereas during the Fall of 2010 the aim was to focus on the electronics and mechanical system which requires the mapping system.

The team developed a simulator program for testing the mapping algorithm, for a future Ceilbot. The algorithm gives a map of the environment using boolean variables where 1 means that there is an object, and 0 a free space. They also work on the renderer, which simulates a 3D laser scanner.

For the mechanical system, the team decided to use the Microsoft sensor Kinect. Kinect was created for XBOX 360 and it is a powerful sensor for motion controlling. This hardware is shown in figure 1.9.

1.3.4 Trunk

The goal of this team was to create the manipulator based on the elephant's trunk. This kind of manipulator is included into the continuum manipulators

(CORPORATION, 2010)

Figure 1.9: Kinect for XBOX 360

and it has a structure able to curve itself, which provides several advantages, such us reaching positions otherwise impossibles, and a better manoeuvrability.

During the Fall 2010, this team worked on the first design of a manipulator controlled by hand and the work will continue in the next semester, testing and automating it.

Design choice

The final design is formed by cables and a multi link axis using extrinsic based actuation. Every section of the manipulator has six joints with an operating angle of 35° each, giving 210° for each section. The manipulator has two sections both independent, and with a length of 396 mm each, so the manipulator is in total almost 800 mm. The first prototype is shown in figure 1.10:

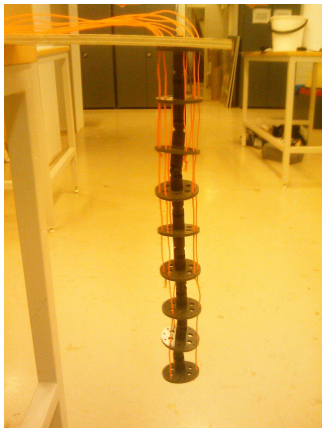


Figure 1.10: First Trunk Prototype

Chapter 2

Hardware and Communications

This chapter explains the design and the implementation, firstly, of the hardware and, secondly, of the communications between the different programs of the robot. One of the aims of the project is to make the communication with GIMnet and MaCI, a communication infrastructure designed for robots. This objective is extensively explained in this chapter.

2.1 Hardware Architecture

During the first weeks of the project, the goal was the design of the Hardware Architecture. To accomplish this, there were several meetings between all the members of the different teams of the Ceilbot Project.

The main idea was to create a flexible and modular architecture, where in future semesters different students would continue the work as easy as possible, even when they did not decide how it would be.

The first decision was to divide all the different systems involved in Ceilbot into different independent modules that would interact and share information. The final modules were the following:

- Camera Module: This module would be in charge of the mapping, and it would use the information of the 3D laser scanner and the kinect camera.

- Motor Controller Module (camera adjustment): Control of the servos to move the camera.
- Motor Controller Module (trolley): Control of the servos and the motor of the trolley.
- Rail Localization Module: Control of all the information of the localization, encoders, landmark system, etc.
- Safety System Module: In charge of the all the security involved in the project.
- Upper Level Module: This module would choose the shortest path, taking into consideration every change in the environment.
- User Interface Module: Graphical user interface for the control of the robot.

After dividing and defining all the modules, the next step was to decide the necessary hardware. The decision was to have two different computers, one placed in the trolley, and the other as a server in a stationary PC. The modules were distributed according to their physical localization and specific features. The final distribution was: in the trolley PC the Camera Module, both Motor Controller Modules (trolley and camera adjustment), the Rail Localization Module and the Safety System Module; while in the stationary PC was the Upper Level Module and the User Interface Module. In figure 2.1 this distribution is shown.

Communication between both PC's would be through wireless LAN, making Internet connection necessary in each PC. For future work it would be really interesting to have access to the User Interface via Internet or mobile phone, so this was taken into consideration for the final communications design.

Attending to the communication between the different programs, the decision was to use MaCI and GIMnet. They provide the communication architecture, and permit all the specific features necessary for this design. The decision to use MaCI means that the Operating System has to be Linux, in every

available version. The decision was to use Ubuntu Linux due to the security of the system and its stability.

The final design of the hardware is presented in figure 2.1:

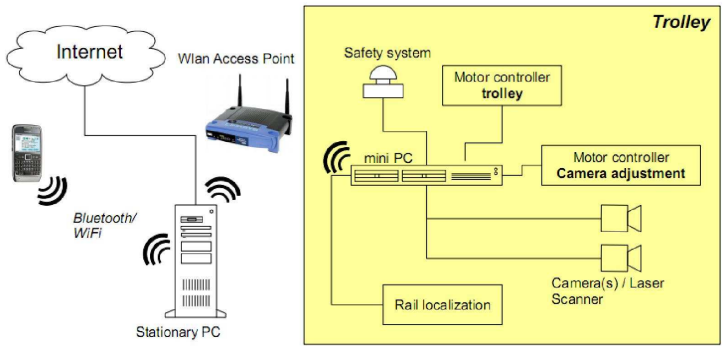


Figure 2.1: Hardware Architecture

2.2 GIMnet

GIMnet is defined by its own creators as a "Communication infrastructure designed for robotics applications. Basically, the system is a remote process communication implementation, which additionally functions as a base architecture for the software system". (Group, 2010b)

GIMnet was designed and created in the Aalto University by the Generic Intelligent Machines Research Group, and it is the basic level of the the communications in Ceilbot. In the core of the infrastructure we can find the hubs. These hubs require Linux to run, and these create a Virtual Private Network after running a program called tcpHub. These hubs allow the communication between the different client modules. It is also allowed the creation of different hubs and connecting them through all the modules. This feature permits a high level of scalability and modularity. Moreover, for GIMnet, the real localization of the module is completely transparent so, if in a future where anyone wants to move one module from a computer to another, this is completely possible. To accomplish this, GIMnet encapsulates the network and gives to every

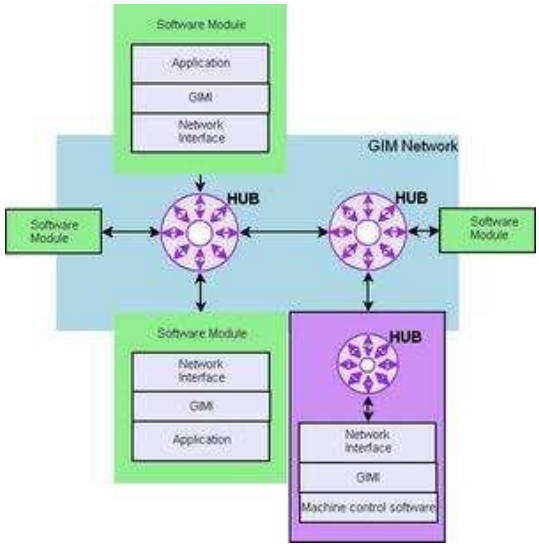


Figure 2.2: GIMnet basic structure

module one name and one ID independently of the hub. The basic structure of GIMnet is shown in figure 2.2.

The main features of GIMnet are: (Group, 2010b)

- Distributed name and ID Service.
- Unicast, multicast, broadcast.
- Synchronized and unsynchronized data transmission.
- Automatic hub-to-hub and client-to-hub reconnect.
- Service registration, subscription and listing.
- Application level ping.

2.2.1 tcpHub

tcpHub is the network layer of GIMnet. Every packet is sent through tcpHub from node to node. When one node is connected to one hub, the system gives it unique identifiers the Hub Identifier and the Node Identifier. As we have already said, the communication between hubs is possible by creating

one big hub, allowing the communication between modules of different hubs. To accomplish this, tcpHub uses TCP/IP for creating among the nodes a connection towards the hub.

2.2.2 GIMI

Also called GIM Interface, it provides different features to the clients such us sending and receiving data or subscribing to other clients. Moreover, everything is completely transparent for the developer of the application which uses GIMnet, allowing him to focus one his code and not on the communications. GIMI is the application-layer communication API. It is possible to download GIMI and the rest of the documentation in the website (Group, 2010b).

2.2.3 Communication

When it is necessary to start a communication between different programs via GIMnet, the first step is to run the tcpHub. To initiate this, it is necessary to open the Linex console, go to tcpHub folder and start it in an available port. In the next example it is possible to see how to run the tcpHub in the port 50000:

./tcpHub -p 50000

After that, every program which starts at the port 50000 has the possibility to communicate with the rest of the programs in the same hub.

2.2.4 GIMnet Implementation

GIMnet is a powerful tool for communications in robotics when it is necessary to send or receive simple information between different modules. For that reason, it is used in the project for the communication between the Graphical User Interface and the Upper Level Software. However, when it is necessary to send information coming from different hardware such us laser scanners or web

cameras, GIMnet could be quite complex because it requires the creation of interfaces between GIMnet and the hardware. In order to solve this problem the Generic Intelligent Machines Research Group had created MaCI, which is based on GIMnet and uses all of its features, and provides the interfaces required by the hardware. For a better understanding of the communication requirements and solutions, the implementation of GIMnet and MaCI is explained at the end of the MaCI section, in the MaCI implementation. (see 2.3.7).

2.3 MaCI

2.3.1 Definition

MaCI (Machine Control Interface) is a software library for robotics based on GIMnet, which provides several interfaces between the hardware and the communication module (GIMnet). The main idea is to help the developers by providing them an already developed and reusable library with all the different modules they could need. Of course, all the features of GIMnet are in MaCI, and the integration between them is automatic.

It can be read in the main website of MaCI (Group, 2010c) that the leading idea of MaCI is to provide:

- Reusable software components for robotic research.
- Define an unified interface for module communication.
- Provide implementation guidelines for the developers.

MaCI was born because sometimes, for the developers, it was quite complex to use GIMnet tp provides "services", and it was necessary to create some specific interfaces. MaCI is a hardware abstraction layer enhanced with some higher level mechanisms which allows it to be used as a control system even for complex problems.

2.3.2 MaCI structure

MaCI is composed of modules. These modules are process where are at least one interface. These interfaces provide data or actions, between the client (service consumer) and the server (service provider). Basically the main structure of every communication in MaCI is the following (Saarinen, 2010b):

- Server or Service Provider.
- Client or Service Provider.
- Data Types.
- Data Container: It encapsulates the data type for the transmission.

The structure of MaCI is based on the idea "divide and conquer" and for that reason it is possible to found five different items: (Saarinen, 2010b)

1. Bus: Directory with several reusable drivers for bus devices. This directory would be used in case that drivers would be needed.
2. Drivers: Directory which consists of a collection of software drivers. The drivers do not depend on any GIMnet or MaCI functionality.
3. Interfaces: MaCI interfaces are the way of communication between modules. It contains all the code related to MaCI services. As we have defined before, it needs: the data type, the data container, the client and the server. MaCI contains several interfaces but it is also possible to build new ones. For a better understanding, figure 2.3 represents how the interfaces work.
4. Modules: It is the union between one interface (at least) and one driver, using any hardware. It is the main "object" of MaCI and it is also reusable.
5. Machine collections: Collection of one or more modules which are connected creating a robot. With the MaCI Launcher it is possible to start everything at the same time. An example of a machine is shown in figure 2.4.

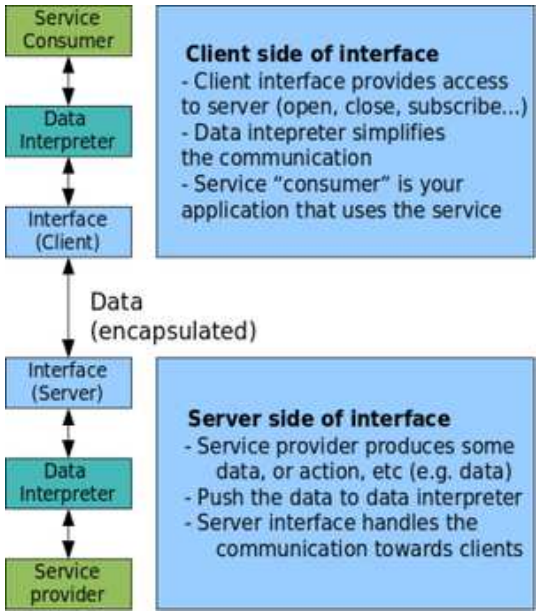


Figure 2.3: Representation of a MaCI interface connection (Saarinen, 2010b)

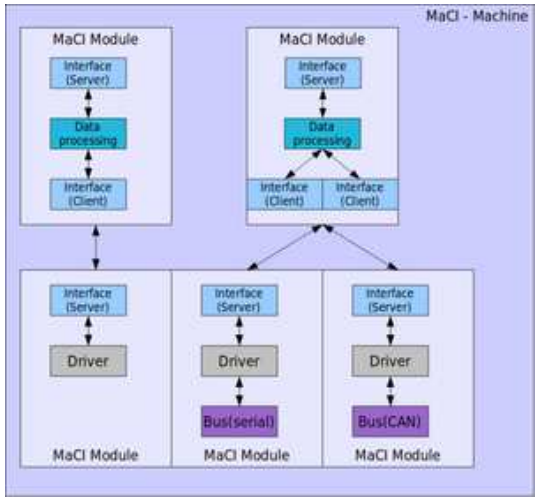


Figure 2.4: Image of a machine composed in MaCI citeproc

2.3.3 MaCI Clients

Different clients exits which can be used in the project. Due to the project being still in the early stages, not all of the final clients which would be used in the future, are in this version. However, to understand all the features of MaCI, and to help in the future work of Ceilbot project, the next lines are a small summary of several clients. In the future work, also MaCI servers would be necessary.

- Image Client: This client provides different functionalities for getting images from the robot. It could be used for web cameras.
- JointGroupCtrlClient: It allows control of the servos of the robot.
- SpeedCtrlClient: For controlling the speed, the angular speed, and the acceleration.
- PositionClient: It gives the position of the robot. It has different odometry options, and parameters.
- BehaviourClient: This client controls the behaviour of the robot. (i.e. it could used for stopping the robot)
- RangingClient: This client allows all the features for laser scanners and bumpers.
- IOClient: It provides the control of input and output signals, analog and digital ones.
- TextClient: Client for sending text to robot.
- SerialClient: Client which provides serial communication to the robot with serial devices.

2.3.4 Gimbo

Gimbo is an user interface for controlling robots. This application allows the checking of all the different services which the machine/robot provides and also

to initialize them. Gimbo is directly connected with MaCI, and it shows all the MaCI clients and servers which are available in the robot. Inside Gimbo exists one tool specially important for this project: MuRo. This application is very powerful: allows the user to control the robot, to check the position of the robot, or to check the battery. In this project MuRo was used to check the viability of the PositionClient. In figure 2.5 it is possible to see this user interface.

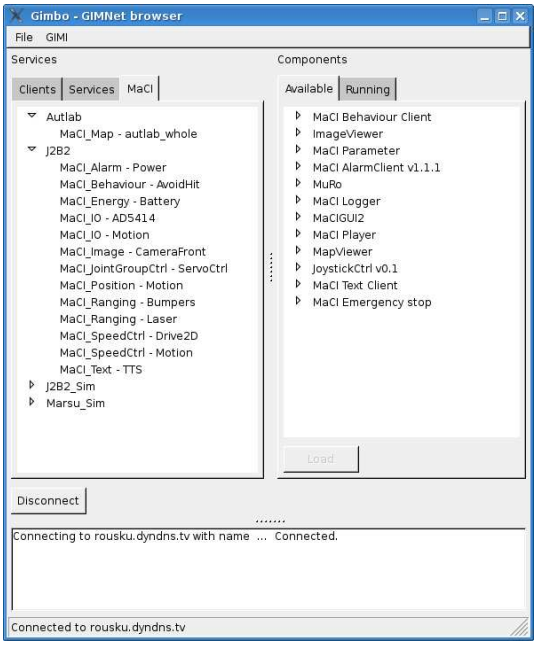


Figure 2.5: MaCI user interface: Gimbo

2.3.5 FSRSim

During the Fall 2010 the aim of the project was to design and build the first Ceilbot prototype. It implies that all the work of the Software Team was made without a working version of Ceilbot. Due to this, the work was done and tested using one simple but really effective simulator created by the Aalto University, the Field Service Robots Simulator (Group, 2010a). This simulator allows to work with simulated robots which have all the features and systems that a real one would.

The simulator has different maps according to the environment you want to simulate, but none of these maps was like our intended environment. For that

reason Antti Maula (expert in MaCI) designed and created one map called "Ceilbotworld". This map is a crossroad which allows to work, firstly, with MaCI and GIMnet, and secondly, with several algorithms necessary for the Upper Level Software. The map simulates the rail-system with corridors which force the robot to move through them as the rails will make in the real system. In figure 2.6 this map is shown.

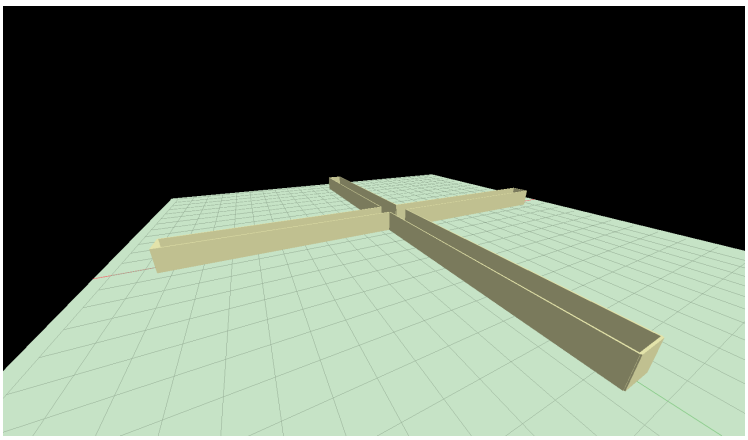


Figure 2.6: Ceilbot World in the FSRSim

The Ceilbot World map was only used in the first steps of the work. Finally, due the lack of some MaCI modules of the robot in it, the final map was changed. This modules were that the robot does not have the next MaCI clients: SpeedCtrlClient and ImageClient, both of them essential for the project, because the Graphical User Interface uses both.

The final map was the "fsr2010.xml". The selection of this map comes from a compromise decision. This map does not represent a rail system, it simulates a house with different rooms, but the robot has complete motion freedom. However, the most important think was to use a map where all the necessary MaCI clients were available, and for that reason the fsr2010 map was the choice. In figure 2.7 is possible to see this map.

2.3.6 MaCI and GIMnet applications

The decision to use of MaCI and GIMnet in the project was not arbitrary. Several projects done in different universities of Finland and Sweden explain

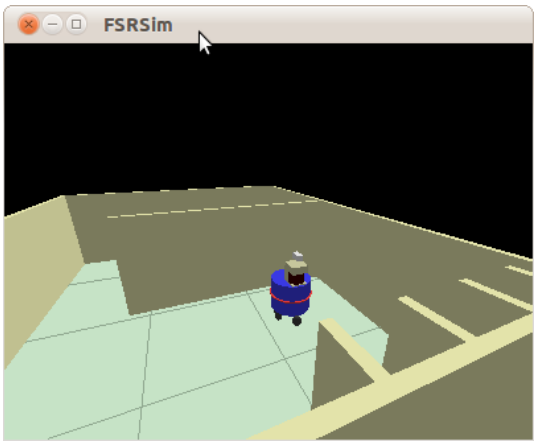


Figure 2.7: Map fsr2010 of the FSRSim

and demonstrate how powerful and flexible they are. The opportunities they provide, create a perfect environment for research projects and, in the future, for industrial projects. The easy way to test and implement small programs or algorithms, the modular design, and the easy implementation of the libraries, even in different robots, are some of the features pointed out by the users. Two projects that use MaCI or GIMnet are explained in the next subsections. (Saarinen, 2010b)

GIMnet on the MICA Wheelchair

MICA (Mobile Internet Connected Assistant) is a wheelchair designed by H. Frediksoon and K. Hyyppä, from LuleåUniversity of Technology, in Sweden. The aim of the project is to research navigation systems which can help the daily life of disabled people.

The project needed a software platform which provided an easily sharing of data between the different programs, and also the possibility of managed through Internet. Moreover, the research group wanted to build a library with different drivers which could be easily reused in other projects. To accomplish this the final decision was to use GIMnet, even when at the beginning of the project it was not used.

Control of work machines with MaCI

MaCI was use for the control of three different compact wheel loaders. This project was done by the Tampere University of Technology in Finland. The result of the project was that MaCI allows the use of the same interfaces for different robots (or in this case work machines), even when the kinematics and their lower-level control are different.

Several MaCI interfaces were used in this project: The CoordinateDrive interface, the EmergencyStop, the Position, the SpeeCtrl and the JoinGroupCtrl. The final conclusion was that all of these interfaces work perfectly with every work machine.

2.3.7 Communication Implementation

Wrapper Classes

Currently, the Ceilbot project needs to use three different MaCI clients and one GIMnet connection. These clients are:

- SpeedControlClient: Used by the GUI for moving the robot.
- PositionClient: Used by the ULS and the GUI, it provides the exact position of the robot.
- ImageClient: Used by the GUI, it gives the camera-streaming.
- GIMnet connection: It provides the communication between the GUI and the ULS and sends the target point.

For the initiation of all these clients, it is necessary to start one gimi object, which represents the connection to the hub. The gimi object is basically the address, the port of the hub, and the name of the service. In the Ceilbot project the default connection parameters are:

- Address hub: asrobo.hut.fi

- Port hub: 50002
- GIMnet sender: ClickPositionSender
- GIMnet receiver: ClickPositionReceiver
- Position Client: FSRSim_J2B2.MaCI_Position.Motion
- Speed Control Client: FSRSim_J2B2.MaCI_SpeedCtrl.Motion
- Image Client: FSRSim_J2B2.MaCI_Image.CameraFront

The wrapper classes are used to simplify the use of listed clients, and they connect the application to the MaCI interfaces. This wrapper classes are completely necessary for the GUI, because they are the middleware framework between the GUI and MaCI. They are also used by the ULS. This is possible because they are in the main folder of the project, which can be used by every application. (Müller, 2010)

It is important to point out that if different service servers want to communicate between them, they need to initialize the same gimi object. For that reason the wrapper classes are implemented as singleton pattern, which allow the MaCI clients to be reused as well as the GIMI connection object.

The figure 2.8 shows the architecture of this wrapper classes.

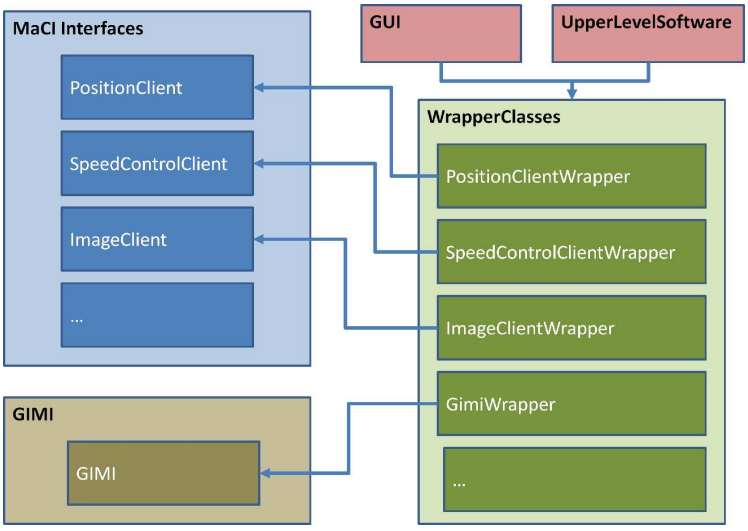


Figure 2.8: Ceilbot Wrapper Classes

The wrapper classes used by the ULS are in the Appendices C and D.

Communication Architecture

At the beginning of the chapter the hardware architecture is explained and also described in the figure 2.1. Taking in consideration this design, the GIMnet and MaCI design in shown in the figure 2.9.

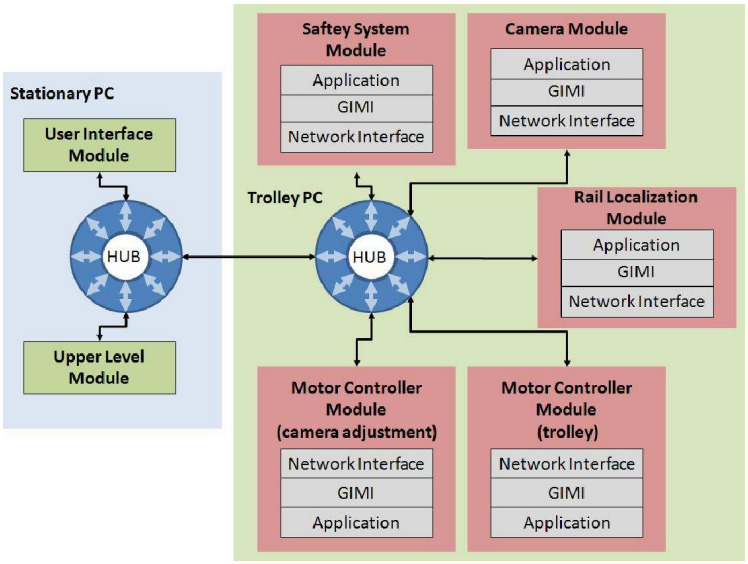


Figure 2.9: Ceilbot GIMnet Implementation

There are two different hubs, one per computer, connected between them for a complete communication. One important feature of GIMnet is that the final distribution is completely transparent for it ,allowing future changes in the design, and working automatically. At this point of the project the Trolley PC are not available and all the work have been done in the Stationary PC, and also simulating the Trolley PC with another laptop.

The ULS requires two different clients, the PositionClient and one GIMnet client. The PositionClient is connected with the FSRSim and it provides the exact position of the robot in the rail system. The GIMnet client is connected with the GUI and it receives the target point. The user clicks on the map situated on the GUI and this information is sent to the ULS.

In the appendix A, there are flowcharts diagrams for a better understanding of

the code corresponding with MaCI and GIMnet. The corresponding diagrams are the figure A.3 and A.4. (the notation A."number", represents that the figures are in the appendix A, and they are the number 3 and 4)

2.4 Chapter Conclusions

MaCI and GIMnet provide a perfect communication environment, allowing the development of modular and scalar programs. This features are esential in testing environments, due to the importance of having an easy and fast implementation of different algorithms. There is one more important charasteristic: the transparence of MaCI. It permits to move every module between hubs working the entire program with no change. Moreover, MaCI and GIMnet allow to work in groups in a parallel way, optimizing the time of the project develop fase.

The implementation of MaCI and GIMnet was a complete success, and the objective of demostrate that it was possible to use them in Ceilbot was reached. Several programs would be added to form the entire Ceilbot, but the base is already running with this work.

Chapter 3

Upper Level Software

The main objective of the project was to create the upper-level software which controls the behaviour of the robot and organizes every plan of the robot, taking in consideration its own features and the environment.

To reach this objective, the project was focused in the implementation of an algorithm for calculating the closest path between two points, the source point and the target point. This algorithm is the Dijkstra's Algorithm. The route of the path, and even the target point, would be different in every case, due to the different shape and weight of the objects around the robot.

Basically, two objectives should be reached by the ULS:

- To determinate the fastest route between the actual position of the robot and the target point.
- To determinate the most efficient target point, according on the weight, and the shape of the object to manipulate.

3.1 Dijkstra's Algorithm

3.1.1 What is Dijkstra's Algorithm?

The Dijkstra's Algorithm, or also called Closest Path Algorithm, determinates the closest path between two different nodes, the source node, and the target node. To accomplish this, it uses a graph with all the possible nodes and the weight that each has to reach one another. The algorithm only works with positive weights and the idea is to minimize the sum of all the weights. The name of this algorithm is because it was created in 1959 by Edsger Wybe Dijkstra, born in Holland in 1930.

Figure 3.1: Edsger Wyde Dijkstra

Several and completely different applications of this algorithm exits such us traffic problems, voice recognition, wire's theory or logistics problems. (J. Hernán Restrepo, 2004)

3.1.2 The algorithm

At the beginning, the algorithm needs to check that all the nodes have a distance associated to the rest of the nodes. The distance between two nodes which are impossible to reach are infinite, and the distance between the node to itself is null. Is important to remark that the distance between the node i

to j is not necesarily the same as j to i. Assuming this, the algorithm has the following steps:

1. All the nodes, with the exception of the source node, have an infinite value. The source node has zero value.
2. To unmark all the nodes and set as initial node the source node.
3. From the current node, check the distance to all the unmarked nodes, and select the minimum from the source node.
4. When one node has the minimum distance is marked and set it as "current node". One marked node is never checked again.
5. If there is not more unmarked nodes the algorithm ends. In case that are still unvisited nodes, the algorithm continues from step three.

Run time of the algorithm

The algorithm uses n-1 iterations (with n as the number of nodes), and in every iteration the algorithm needs n-1 or less comparations. After that, is necessary to make one addition, and one comparation for refreshing the mark of the nodes. In total, for each iteration is necessary 2(n-1) operations.(Seth Pettie, 2005) It is important to remark that the number of iterations on the algorithm is independent of the distance between the nodes, it is only function of the number of nodes. Due to the number of the iterations is n-1, the result is the next theorem:

Theorem 1 *The Dijkstra's Algorithm makes $O(n^2)$ operations for the determination of the closest path between two nodes of a ponderate simple graph, conected, and not directed, with n nodes.*

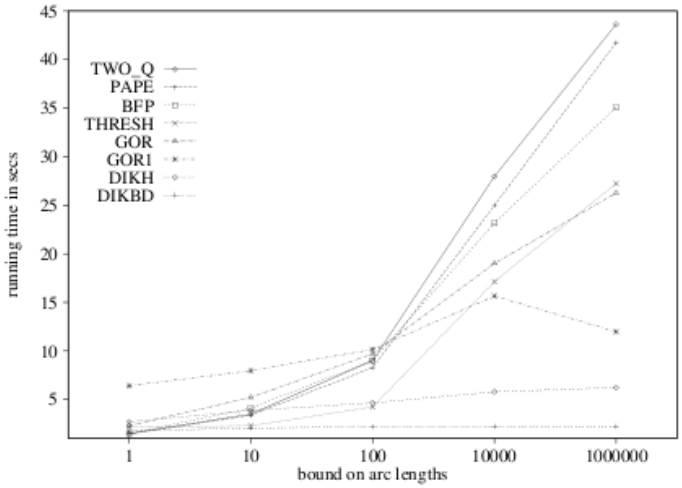
3.1.3 Comparative with other algorithms

It exists different classes of shortest path problems. The selection between one algorithm or another are going to be based on the specific characteristics of

the problem to solve.

In our case all the weights are positive due to that they are based on the real distance between nodes. Other different approach of our problem could be to define the weights in function of the time that the robot needs to reach it, but also in this case the weights are positive. In this case of problems the best algorithm is the Dijkstra's Algorithm, which is robust, and the implementation is quite competitive. In case that the problem needs negative weights the best algorithm is the Bellman-Ford. (Boris V. Cherkassky and Radzik, 1993)

In the figure 3.2 is possible to visualize one experiment between different algorithms in a problem with acycling networks. This experiment is a good example about why the Dijkstra's Algorithm is the best option due to the acyclic networks never have negative weights, so is basically our problem. The experiment has been made with 1310702 nodes and 524288 arcs.



$[L, U]$	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO_Q	THRESH
[1, 1]	**1.50**	**2.31**	**6.43**	**2.71**	**1.77**	**1.45**	**1.49**	**2.16**
	1.00	1.61	4.48	1.00	1.00	1.00	1.00	1.00
[0, 10]	**4.12**	**5.22**	**7.97**	**3.88**	**2.07**	**3.39**	**3.50**	**2.30**
	2.67	4.36	5.94	1.00	1.00	2.85	2.84	1.01
[0, 100]	**9.03**	**9.71**	**10.16**	**4.66**	**2.18**	**8.29**	**8.99**	**4.24**
	6.15	8.93	8.21	1.00	1.00	8.07	8.05	1.78
[0, 10000]	**23.17**	**19.01**	**15.67**	**5.80**	**2.23**	**24.99**	**27.97**	**17.12**
	16.75	19.48	14.15	1.00	1.00	27.31	26.07	9.16
[0, 1000000]	**35.09**	**26.27**	**11.99**	**6.23**	**2.23**	**41.76**	**43.64**	**27.25**
	26.77	27.96	12.26	1.00	1.00	47.60	41.89	16.03

Figure 3.2: Comparation of the running time in acycling networks on different pathing algorithm (Boris V. Cherkassky and Radzik, 1993)

In the figure 3.2 two different approaches of the Dijkstra's Algorithm [DIKH;

DIKBD] are compared with six more algorithms. The differences between both approaches are that DIKBD uses bucket data structure, whereas DIKH is basically the simple Dijkstra. When the distance of a node changes, the node is moved from a bucket corresponding to its old distance label and serted into the bucket corresponding to the new one. The DIKBD is a good approach specially for problems with negative number, otherwise the differences between both are not so high. Taking in consideration that the implementation of the DIKBD is more complex, the final decision was to use the DIKH.

3.1.4 ULS Implementation

The implementation of the Dijkstra's algorithm was made in C++ and it uses one adjacency matrix to do it. For helping the understanding of the algorithm the cross-road rail system is going to be used as example.

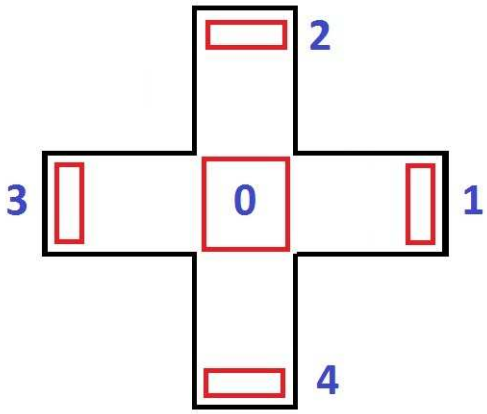


Figure 3.3: Cross-road Rail System

The system has five different elements, one per element of the rail system. This way to divide the system is because one possible change in the system could be that one part of the rails breaks. It is important that the program realize this change, and avoid this rail element. The elements are in colour blue in the figure 3.3.

Every element of the rail has one "beginning point" marked in red in the figure 3.3. This points are the points which are going to be in the adjacency matrix,

and for that reason the nodes of the algorithm. Obviuosly the final point and the source point are also going to be in the matrix.

The code starts reading one text file called "Matrix" where are the main information of the system, the adjacency matrix of the static elements (without final and source point). In our example, assuming a length of every element of 3 meters, the matrix would be like the following:

$$\begin{bmatrix} 0 & 3 & 3 & 3 & 3 \\ 3 & 0 & \infty & \infty & \infty \\ 3 & \infty & 0 & \infty & \infty \\ 3 & \infty & \infty & 0 & \infty \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \tag{3.1}$$

In the matrix 3.1 is easy to check how to value is corresponding with the basic rules (see 3.1.2) of the Dijkstra's Algorithm in our example. After reading this file, the system adds two more columns and rows, the source ones and the final ones. The next step is to check in what element the source and the final point are. We are going to assume the situation of the figure 3.4.

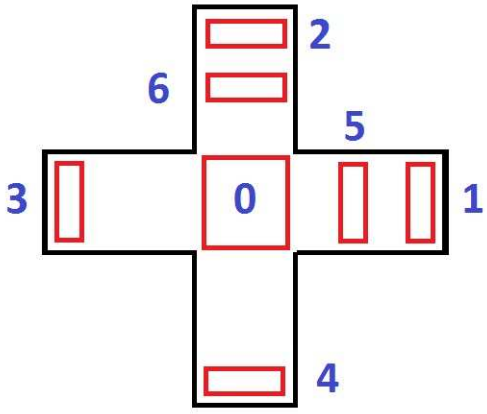


Figure 3.4: Cross-road Rail System after adding the final and source point

After checking the element of both nodes, the code would copy the column of the element where the final and the source point are, to the column and the row of the target and source, respectively. The adjacency matrix in this case would be now like the matrix 3.2

$$\begin{bmatrix} 0 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & \infty & \infty & \infty & 0 & \infty \\ 3 & \infty & 0 & \infty & \infty & \infty & 0 \\ 3 & \infty & \infty & 0 & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty & 0 & \infty & \infty \\ 3 & 0 & \infty & \infty & \infty & - & - \\ 3 & \infty & 0 & \infty & \infty & - & - \end{bmatrix} \tag{3.2}$$

The last step is to change some values on the "new" rows and columns, and to complete the matrix. For the first step it could be two different options:

1. The weights which were infinity, are still going to be infinity. The nodes which were impossible to reach from the "original element" node, are still going to be impossible to reach.
2. The rest of the weights are going to be refreshed in function of the distance between the node and the "original element" node.

In order to complete the matrix, there exists also two different options:

1. The final node and the source node are in the same rail element. The weight between them will be directly the rest.
2. The final node and the source node are in different rail elements. The weight between both nodes will be infinity.

In both cases the weight between one node and itself will be always zero.

With all these different cases, the final matrix is shown in the figure 3.3.

$$\begin{bmatrix} 0 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & \infty & \infty & \infty & 0 & \infty \\ 3 & \infty & 0 & \infty & \infty & \infty & 0 \\ 3 & \infty & \infty & 0 & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty & 0 & \infty & \infty \\ 3 & 0 & \infty & \infty & \infty & 0 & \infty \\ 3 & \infty & 0 & \infty & \infty & \infty & 0 \end{bmatrix} \tag{3.3}$$

At this moment, the matrix is complete and is possible to start the Dijkstra's Algorithm. The algorithm uses is a modification of the one made by Vinodh Kumar B.(B., 2010). The code corresponding to Adjacency Matrix and Dijkstra's Algorithm follows the diagrams of the figures A.9, A.10, A.11 and A.12 (This figures are in the appendix A) . The entire code is in the appendix B (see addtomatrix and Dijkstra's objects).

3.2 Behaviour of the Robot

3.2.1 Behaviour Theory

For understanding the different approaches of the behaviour robot theory is necessary to think in a robot as an agent. The definition of agent is a system, in an environment, which is able to take decisions by itself according to its own design. (R.R, 2010) Inside this definition three main words appears which explains what is the robot behaviour. This words are: system, environment and decision. Being more specific (Nehmzow, 2009):

- The program running on the robot (the decision or task)
- The physical makeup of the robot (sensors, motors, etc which are part of the robot)
- The environment itself.

To solve the interaction of the three different elements of the figure 3.5 is the aim of the Behaviour Theory. Several approaches have been done to solve it but only three of them are remarkable enough:

- Deliberative Control.
- Reactive Control.
- Hybrid Control.

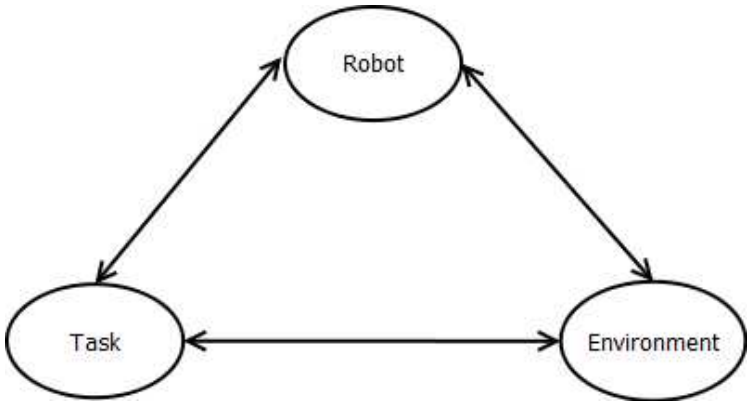


Figure 3.5: Basic elements of Robot's Behaviour

Deliberative Control

This control is based on the hierarchy paradigm which is born from an introspective vision of how humans think. The algorithm is sequential and ordered. It has three steps: Sense, Plan, and Act.

The robots built before the 90's used this approach ,and this solutions were developed just for specific applications instead of thinking in generic global architectures which allowed to be used by other applications. These robots were slow because the planning requires to search, and to search requires an world model. (Vainio, 2007)

Reactive Control

Behind the theory of the Reactive Control is the aim of modelling the environment and to react to every possible situation directly without thinking. The planning is avoided obtaining a faster model than the Deliberative one. These feature is the main one, this agents are fast and efficient, and normally, very robust (R., 1986). Therefore, this control has only two steps: Sense and Act. The main characteristics of this control are the following:

- The execution is very fast, so they work in real time.
- No memory is used, are based on "stimulus-response".

- The behaviour of the robot is able to adapt to the changes of the world.

Hybrid Control

This way of control combines the two before theories. Firstly, the agent plans how to divided the task in several subtasks, and how it should act. After that, it executes the corresponding behaviour of each subtask. This approach allows to control the robot in real time, using asyncronuos processing where the deliberative modules are working independently of the reactive modules. (de Computación, 2010)

3.2.2 Study of the behaviour in a robot mounted on a rail system

The main objective of the ULS is to find the closest path between two points: the source point and the final point. The source point is the real position of the robot and the final point is the position on the rail system that the robot needs to manipulate the object required by the user. This final position will not be the always the same, changing in function of the shape and the weight of the selected object. There is a third point: the target point. The target point is the point where is the object to manipulate.

One example is going to help us to understand it. If the user ask the Ceilbot to help moving one table, Ceilbot needs to move a heavy object, and for that reason needs to minimize the bending moment, trying to be as close as possible. The name of this situation in the code is "Heavy". However, if the selected object is a pillow, Ceilbot does not need to be worry about the bending moment, and the most important factor would be the time that it is necessary to reach the final point. According with this, the point to reach would be the optimal one instead of the closest one. The optimal point is the faster point to be reached by Ceilbot which allows to get the object. This situation is defined as "Light".

It could be that some points would not possibly be reached by Ceilbot because

the length of the manipulator is less than the smallest distance to it. This case should be avoided by the design of the rail system but in case that it occurs, Ceilbot would give an error to the user.

This is the situation that the ULS solves. According with the theory explained, the behaviour of Ceilbot would be inside the reactive control (see 3.2.1). The Ceilbot reacts directly to the order made by the user, checking the "shape and weight" of the object. Currently, this checking is simulated, asking via console to the user, but in the final version this question would be answered by the mapping algorithm.

Why the control is reactive? Analyzing the problem is easy to check that the characteristics of it makes that the reactive control is appropriate due to the fast reaction of the Ceilbot, the easy implementation, and the adaption of the changes in the environment. However, taking in consideration future additions to the program and studying them, the program has been designed as modular and scalar as possible. When the problems are increasing, and the number of variables are also increasing the best approach is the hybrid control (see 3.2.1) and that is why all the work has been made dividing the code in different modules. The future implementation of the hybrid control must be as easy as possible. Some of this possible future additions could be changes in the path planning due to human interaction or broken parts of the rail system.

3.2.3 Implementation

In this subsection the whole upper-level software is going to be explained with special attention to the implementation of the robot's behaviour, leaving out the explanation of the Dijkstra Algorithm, which would be only referenced (see 3.1).

The ULS starts opening three necessary text files: system, matrix and elements. This files have all the information of the rail system, the system file has all the points, the matrix file the adjacency matrix (see 3.1.4), and the elements file the position of the first point of each element. Everything is saved in variables for a faster run time of the program, opening the files only one time.

In figure A.2 (This notation means that this figure is the number 2, appendix A) is shown this module of the program called "open".

After this, the program enter in a while loop which is repeating until the user decided to quit. The first step inside the loop is to start the GIMnet communication, and wait for one order from the GUI (see 4.1). This order send the position of the object that is necessary to manipulate. This GIMnet communication is explained in the figure A.3. After knowing where is the object to manipulate, the MaCI Position Client is opened and it receives the actual position of the robot in Cartesian coordinates. The Position Server in this case is in the FSRSim (see 2.3.5). This is also explained in figure A.4. It is necessary to know what is this point in the rail system. To do that, each distance between the coordinates received, and all the points of the rail system is calculated. The point with minimum distances will be the selected one. This is also explained in figure A.5.

At this moment, the program has all the information about the environment and only needs to define the object to manipulate. There are two possible definitions: heavy or light. The program asks this to the user by the Linux console (see A.6). In function to the answer the program would open the "Heavy" C++ object, looking for the closest point (see figure A.7), or the "Light" C++ object, looking for the optimal point (see A.8).

To calculate the heavy point, the code find the list of possible points with the minimum distance between the final point and the final point in the rail system. To decide which is going to be the final point, the Dijkstra's Algorithm would be made to all the list of possible points, and the point with the shortest path between the source point and it, would be final point. Nevertheless, to calculate the light point, the Dijkstra's Algorithm would be made to every point in the rail system with a distance between it and the target point minor than the manipulator's length. The final point would be the faster point to reach. Finally, the program goes to the while loop waiting for a new order from the GUI. All the flowchart's diagrams corresponding with the ULS are in the appendix A.

3.3 Chapter Conclusion

The Upper-Level Software is the brain of a robot. Ceilbot ULS at this moment is working on two taks: to determinate the shortest path, and the control of this path according of the shape and weight of the object to manipulate. The actual Ceilbot is able to decide in function of a parameter gived by the user (simulating the decision of the shape and weight of the object that will be done in future by the mapping algorithm) and calculate the shortest path.

In this version of the ULS, the two main tasks are successfully reached, but the work with the ULS is not already finished, and during the next semesters must be completed. Every single possibility that might occur around Ceilbot should be studied, completing a fully and final version of the ULS. Moreover, the actual work has been done with the Field Service Robots Simulator, whereas the final version must work with the real Ceilbot hardware.

Chapter 4

Results and Tests

The tests were done for checking if the three main objectives were reached (see chapter 1). The system used for the test were two laptops. On the first one, the Graphical User Interface (see 4.1) was running, and on the other one, the Upper Level Software (see 3) and the FSRSim (see 2.3.5). For the communication between both systems by MaCI, the parameters were the default ones, connecting to "asrobo.hut.fi" (see 2.3.7). The map used in the test was the crossroad (see 3.3). Before the tests, the GUI is going to be explained.

4.1 Graphical User Interface

The aim of the rest of the Ceilbot Software Team was to develop an easy and extensible user interface which can send orders to the robot, and receives information from Ceilbot's sensing. The first step was to decide the design and the basic element which would form the GUI. The GUI design can be found in the figure 4.1. The GUI is formed by:

- Two video streaming cameras which shown the video of the web cameras or the Kinect (see 1.3.3).
- One map of the system where the user could click every point and select the objective. The orange big dot represents this target point and the

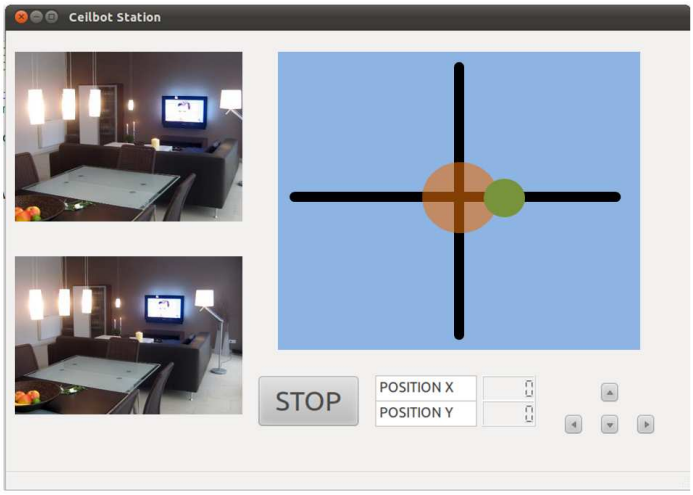


Figure 4.1: First Ceilbot GUI Design

green one the actual position of the robot.

- Four arrows to control the trolley on the rail system.
- Two displays which shown the actual position of the robot.
- An emergency button which stops the system.

For the implementation of the GUI, the development environment selected was Qt Framework, a powerful tool which allows to create it, and connect it with MaCI. Moreover, taking in consideration the possible future work with Qt is possible to develop Internet applications. Qt language is C++.

For this design three different MaCI clients and one GIMnet communication are necessary: SpeedControlClient, PositionClient, ImageClient, and GIMnet for the communication between the GUI and the ULS. The final available version of the GUI ,called Ceilbot Station, is shown in the figure 4.2. More information about the design and develop of the GUI is extensively explained in Klaus Müller Final Report. (Müller, 2010)

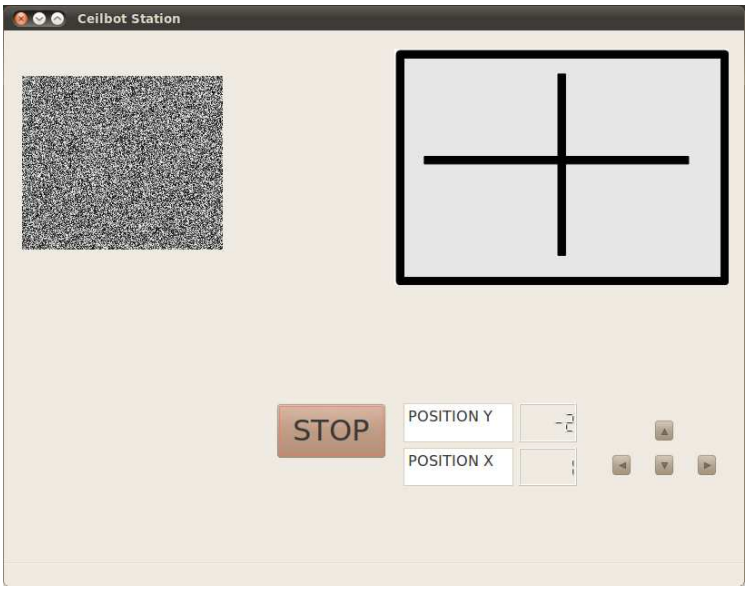


Figure 4.2: Ceilbot Station. View of the GUI.

4.2 Final Tests

In this section a test example is going to be shown and explained. Two different test options are going to be made, one with a Heavy Target and the other one with a Light Target (see 3.2.2).

At the beginning, the program is waiting for a click from the GUI. After this click the coordinates received are shown on the console. Also is possible to see that: the message received is a Ceilbot, and the size of the message is eight bytes. This information is just for checking that the information received is correct. All this information is shown in the figure 4.3.

```
Message is a Ceilbot.
It has 8 bytes of data.
Ceilbot packet: The number that I have received is.... 2.700000..
Ceilbot packet: The number that I have received is.... -1.400000..
```

Figure 4.3: ULS starting. The user has already sended the order

The object that the user want to take is in the position: x= 2.7 y=-1.4, as is possible to see in the last figure. For understanding the actual situation around Ceilbot, the figure 4.4 explains the coordinates of every point in the map.

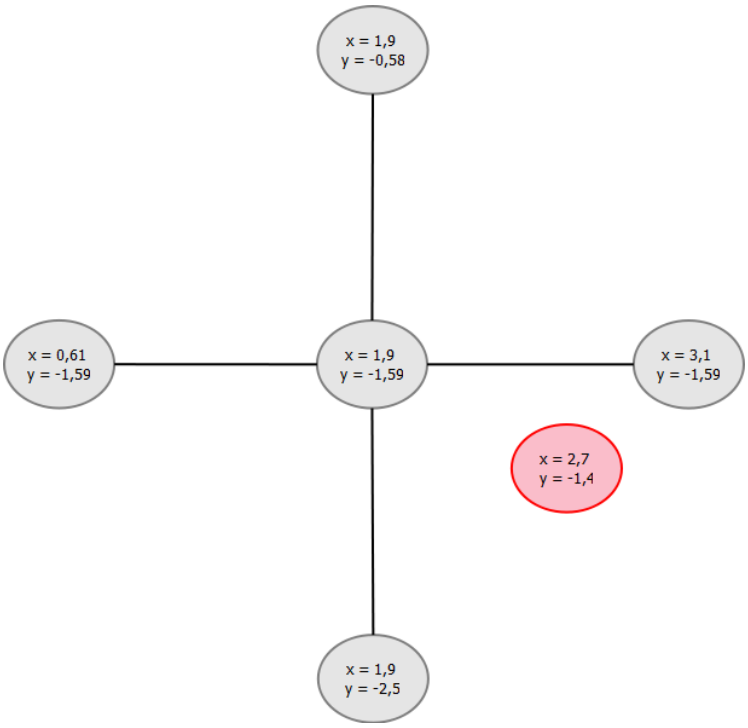


Figure 4.4: Object Point readed from the Position Client.

After this, the Position Client starts and checks on the FSRSim what is the actual position of the robot. In the figure 4.5 is possible to check that the position is x= 0.79 y=-1.58. An small error is checked on this step because the real y-coordinate should be -1,59 but is not important due to is just because on the simulator is impossible to represent the rail system. The actual situation is also shown on the map of the figure 4.6.

```
PositionClient - is loading
0.795568   -1.58231
Select between light object (0) or heavy object (1)
```

Figure 4.5: View of the Position Client in the Linux Console

After this point the program asks if the object that must be manipulated is heavy or light. The difference between this two options is explained in the section 3.2.2. In the final version, when the Ceilbot would be working, this question should not be asked by the program, and with the information of the mapping algorithm it would know what to do.

In case that the user selects the light option, the program has two different points as option in this test. The final selection would be the optimal one, in

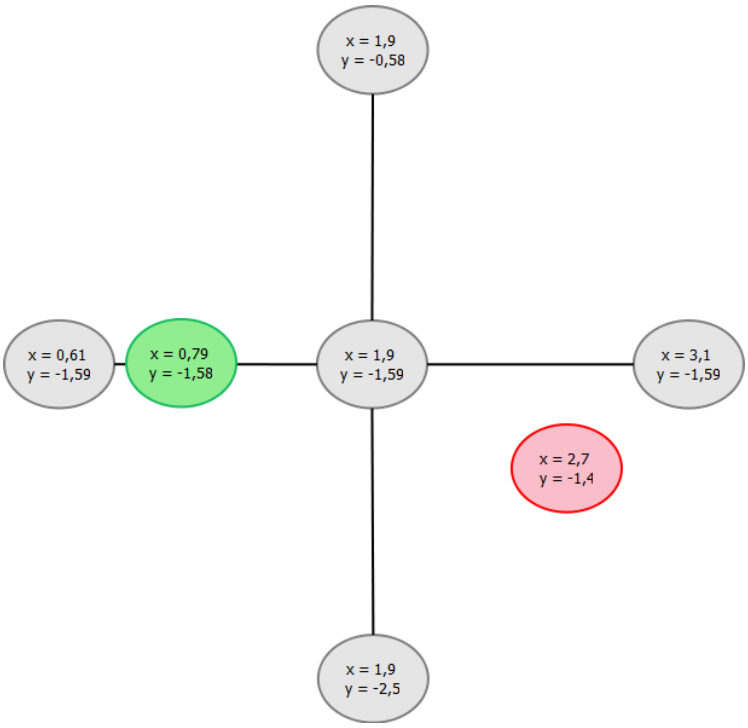


Figure 4.6: Trolley Position readed and added.

this case the second one. The console gives us the distance of every "beginning point" of each element to the target, and finally shows the path. Firstly the zero node, the cross-road, and secondly and finally, the fifth node. The console of the light option is shown in figure 4.7 and the map in figure 4.8.

In case that the selected option would be the heavy one, the algorithm is more restricted, and in this case the final point is x=2,7 y=-1.59, being this point the only option. The heavy option always select the closest one, and in this case is clearly this point. The predecessors are the same ones that in the light one, however, the point five has different coordinates, so is not a problem.

It is important to point out that always the final predecessor must be the number of the "final point". The console of the heavy option and the map situation are shown in the figure 4.9 and 4.10.

```
Select between light object (0) or heavy object (1)
0
The position; X 0.795568
The position; Y  -1.58231
The distance of the position is..0.1806..in the element..4
The targetX is  2.7
The targetY is  -1.4
Starting checking the possible points
The distance of the target is..0.852.in the element.1
The target point is..2.248 -1.59
The distance from 0 is..1.1094
The distance from 1 is..2.3094
The distance from 2 is..2.0194
The distance from 3 is..2.1194
The distance from 4 is..0.1806
The distance from 5 is..0
The distance from 6 is..1.6134
The distance of the target is..0.84.in the element.1
The target point is..2.26 -1.59
The distance from 0 is..1.1094
The distance from 1 is..2.3094
The distance from 2 is..2.0194
The distance from 3 is..2.1194
The distance from 4 is..0.1806
The distance from 5 is..0
The distance from 6 is..1.5894
The predecessor number..1..is..0
The predecessor number..2..is..5
[01] 06:03:40.443, uls.cpp:Gimnet():563; Successfully connected.
```

Figure 4.7: Linux Console corresponding of a Light Target

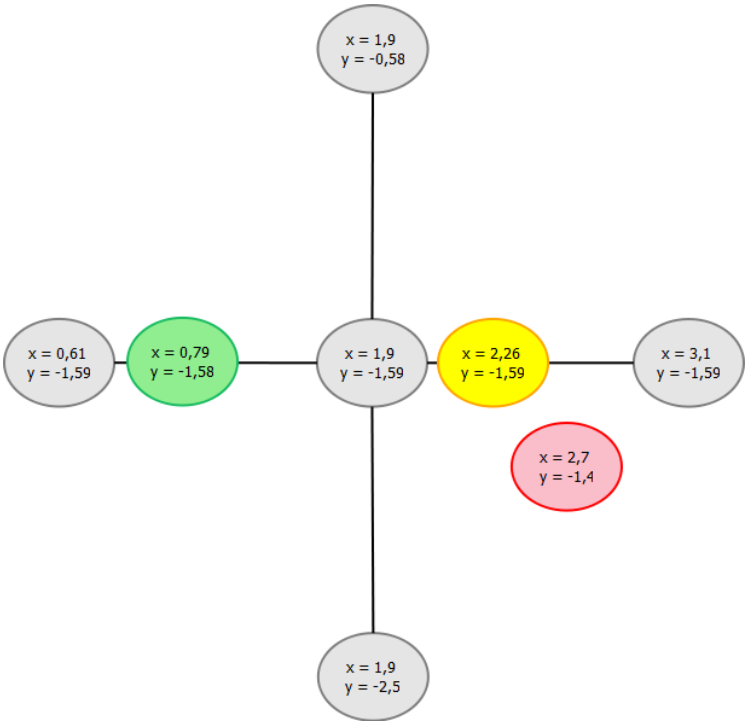


Figure 4.8: Map which explians the "Light Target" situation where the final point is already selected

```
Select between light object (0) or heavy object (1)
1
The position; X 0.795568
The position; Y  -1.58231
The distance of the position is..0.1806..in the element..4
The targetX is  2.7
The targetY is  -1.4
Starting checking the possible points
Found point with minimun selected distance
The distance of the target is..0.396.in the element.1
The target point is..2.704 -1.59
The distance from 0 is..1.1094
The distance from 1 is..1.9134
The distance from 2 is..2.0194
The distance from 3 is..2.1194
The distance from 4 is..0.1806
The distance from 5 is..0
The distance from 6 is..1.5174
The predecessor number..1..is..0
The predecessor number..2..is..5
[01] 06:04:12.284, uls.cpp:Gimnet():563; Successfully connected.
```

Figure 4.9: Linux Console corresponding of a Heavy Target

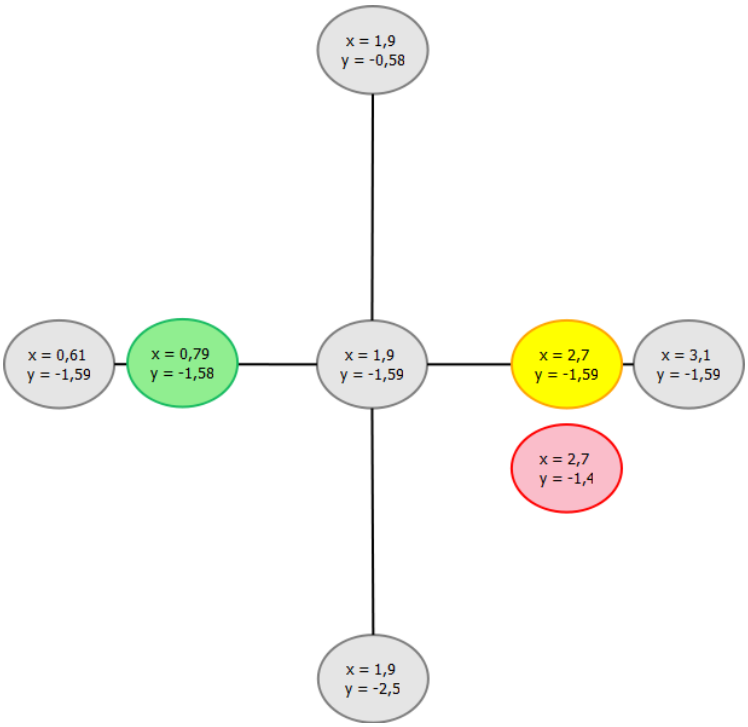


Figure 4.10: Map which explians the "Heavy Target" situation where the final point is already selected

Chapter 5

Conclusions and Future Work

The work of this Thesis is going to help to make the implementation and the testing of the first Ceilbot Home Prototype as easy and fast as possible. All the work has been done without this prototype so several changes would be necessary but the fact that part of the work is already done, made it easier.

All the different parts of this ULS have been designed taking in consideration two main principles: modular architecture, and easy implementation in different environments. The first one is related with adding new features or changing any, trying to make them as easy as possible, avoiding to change parts of the code which are not directly affected. The second one, is about the fact that the Ceilbot will have different environments, and the work made for one, should be the base of the rest of them.

However, the fact of working without the real robot is a problem due to all the work is done by a simulator with an ideal environment. The future work should be focused on the implementation of the different programs on the Trolley PC. Moreover, the ULS must interact with the Mapping Algorithm, the Trolley Module and the Odometry. At this moment some of this requirements are simulated by the FSRSim or by the Linux Console. After this implementation several applications should be added to the ULS such us a variable Adjacency Matrix which changes according with rail elements broken or the detection of humans.

This Thesis also demostrates that MaCI and GIMnet are a perfect communication infrastructure. All the mentioned characteristics of them, creates an ideal tool to develop prototype robots. The easy implementation of them in a program, allows to divide the robot in different modules, increasing the time of the develep and test fases.

References

Arkin, R.C. (1998). *Behaviour-Based Robotics.*

B., V.K. (2010). *Dijkstra's single source shortest path algorithm.*
URL: *http://ds4beginners.wordpress.com/2006/09/17/dijkstras-algorithm/*

Boris V. Cherkassky, A.V.G. and Radzik, T. (1993). *Shortest Paths Algorithms: Theory and Experimental Evaluation.*

Ceilbot (2010). *The Ceilbot project.*
URL: *http://autsys.tkk./en/ceilbot*

CORPORATION, M. (2010).
URL: *http://www.xbox.com/en-gb/kinect*

de Computación, I. (2010). *Paradigmas en Robótica.*

Group, G.I.M.R. (2010a). *FSRSim.*
URL: *http://automation.tkk.fi/AS-84-3144-ProjectWork/Tutorial*

Group, G.I.M.R. (2010b). *GIMnet.*
URL: *http://gim.tkk.fi/GIMnet*

Group, G.I.M.R. (2010c). *MaCI.*
URL: *http://gim.tkk.fi/MaCI*

J. Hernán Restrepo, J.J.S. (2004). *Aplicación de la teoría de grafos y el algoritmo de Dijkstra para determinar las distancias y las rutas más cortas.*

J.Borenstein, H. and L.Feng (1996). *Where am I?, Sensors and Methods for Mobile Robot Positioning.* University of Michigan.

Müller, K. (2010). *Ceilbot Software Group. Final Report.*

Nehmzow, U. (2009). *Robot Behaviour: Design, Description, Analysis and Modelling.*

R., B. (1986). *A Robust layered control system for a mobile robot.*

R.R, M. (2010). *An Introduction to AI Robotics (Intelligent Robotics and Autonomous Agents).* MIT Press.

Saarinen, J. (2009). *The J2B2 Experiment.*

Saarinen, J. (2010a). *Odometry Estimator for MaCI using EKF.*

Saarinen, J. (2010b). *Proceedings of GIMnet 2010.* Aalto University, Center of Excellence in Generic Intelligent Machines.

Saarinen, J. (2010c). *Recycling Robotics, Garbage collecting Robot Application.*

Seth Pettie, V.R. (2005). *A shortest path algorithm for real-weighted undirected graphs.*

Torrubia, G.S. and Lozano, V.L.T. (2001). *Algoritmo de Dijkstra. Un tutorial interactivo.* University Politénica de Madrid, Faculty of Computer Science.

Vainio, M. (2007). *Behaviour-based robotics.* Automation Technology Lab / HUT.

Appendix A

Flowcharts Diagrams

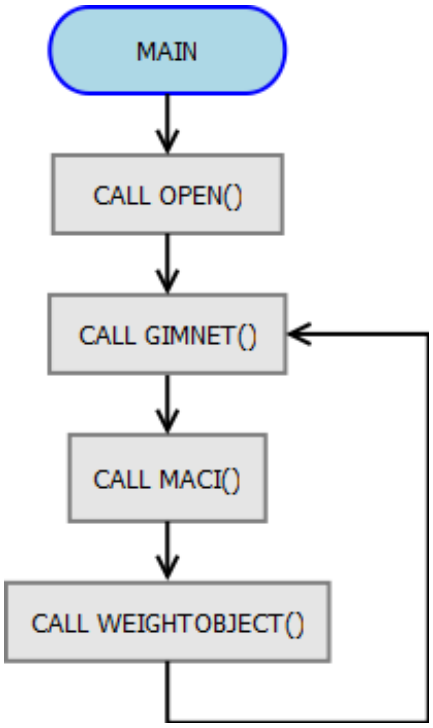


Figure A.1: Main Ceilbot ULS's program diagram

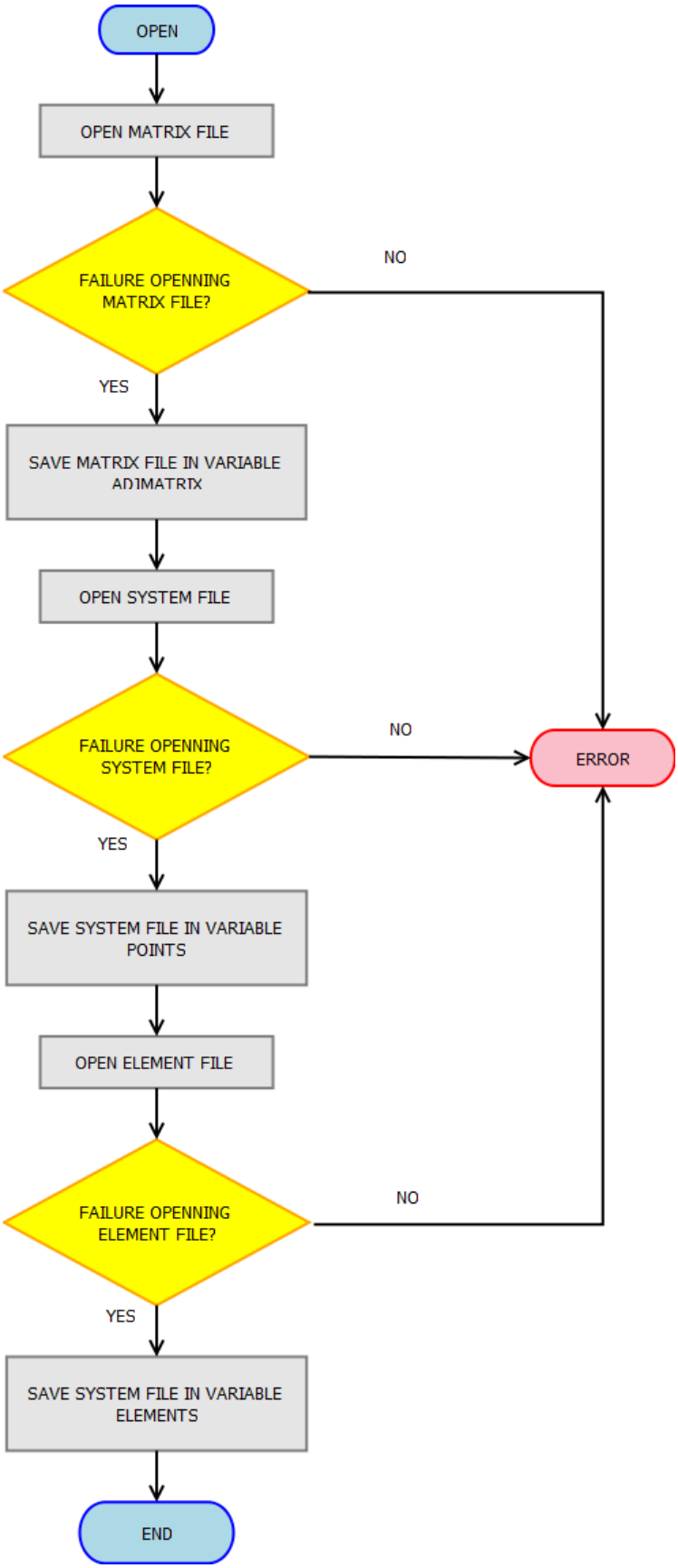


Figure A.2: Open Diagram. The three text files where are all the information of the envinment are opened and saved

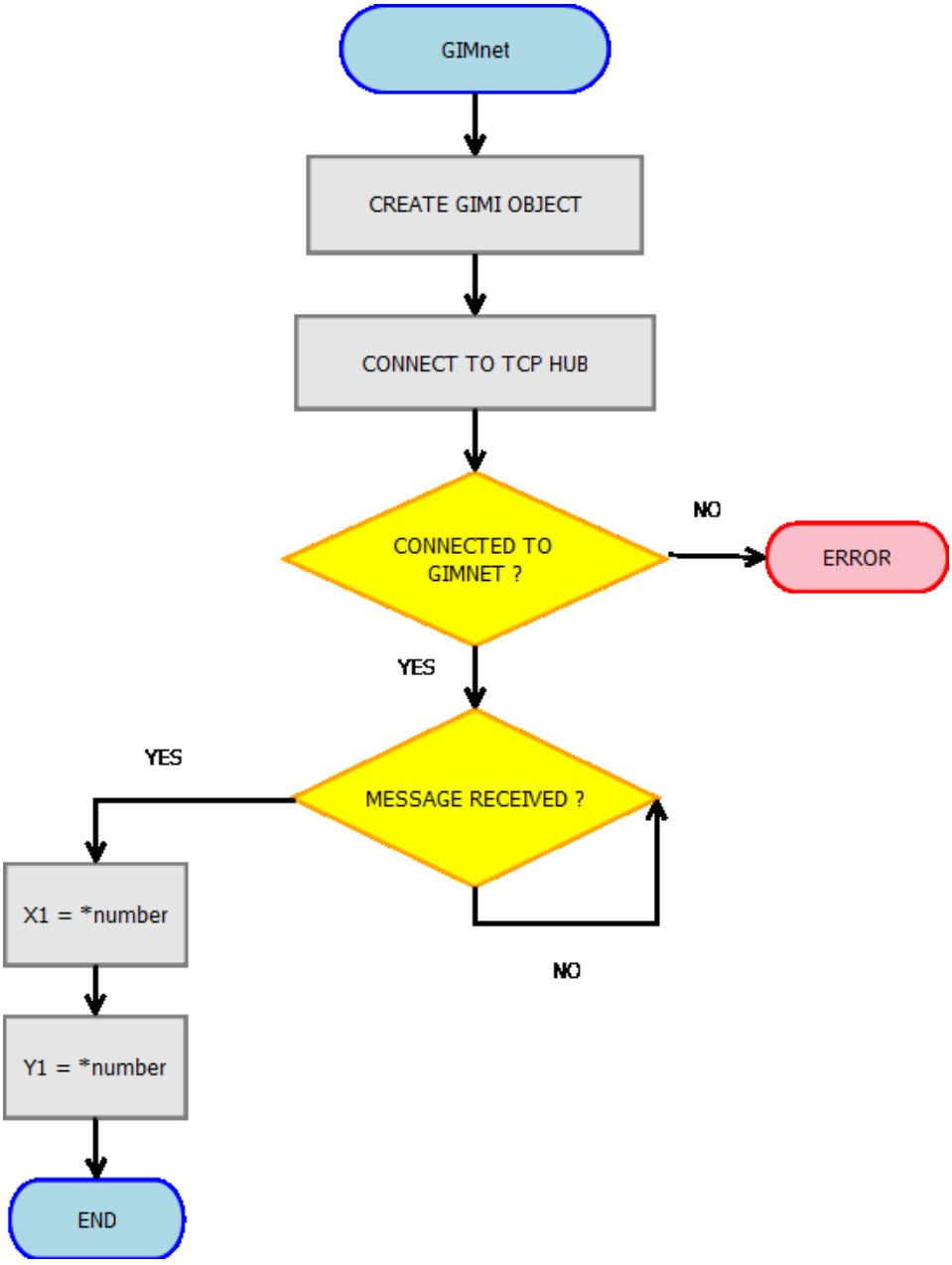


Figure A.3: GIMnet code diagram. It represents the openning of the GIMnet client to receive the order from the GUI 4.1

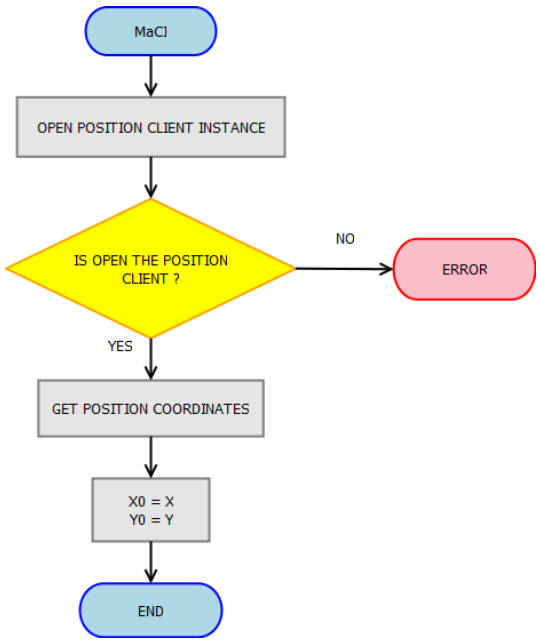


Figure A.4: MaCI code diagram. It opens the MaCI Position Client to receive the position of Ceilbot

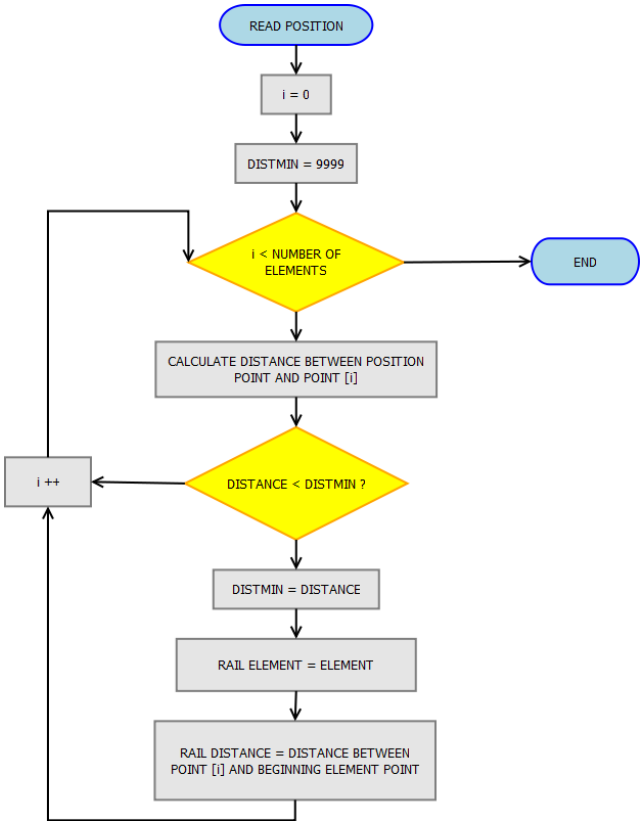


Figure A.5: Read Position Diagram. The code finds the point in the rail system where Ceilbot is

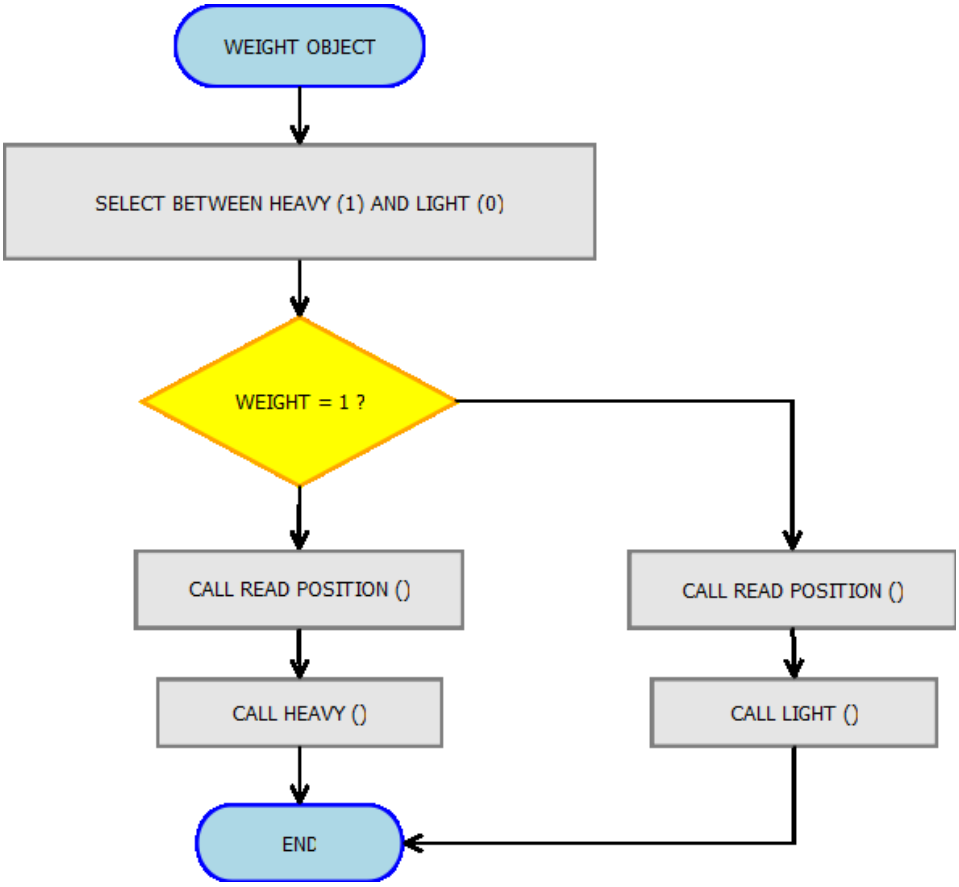


Figure A.6: Weight Object Diagram. It simulates the decision of the shape and weight of the object to manipulate

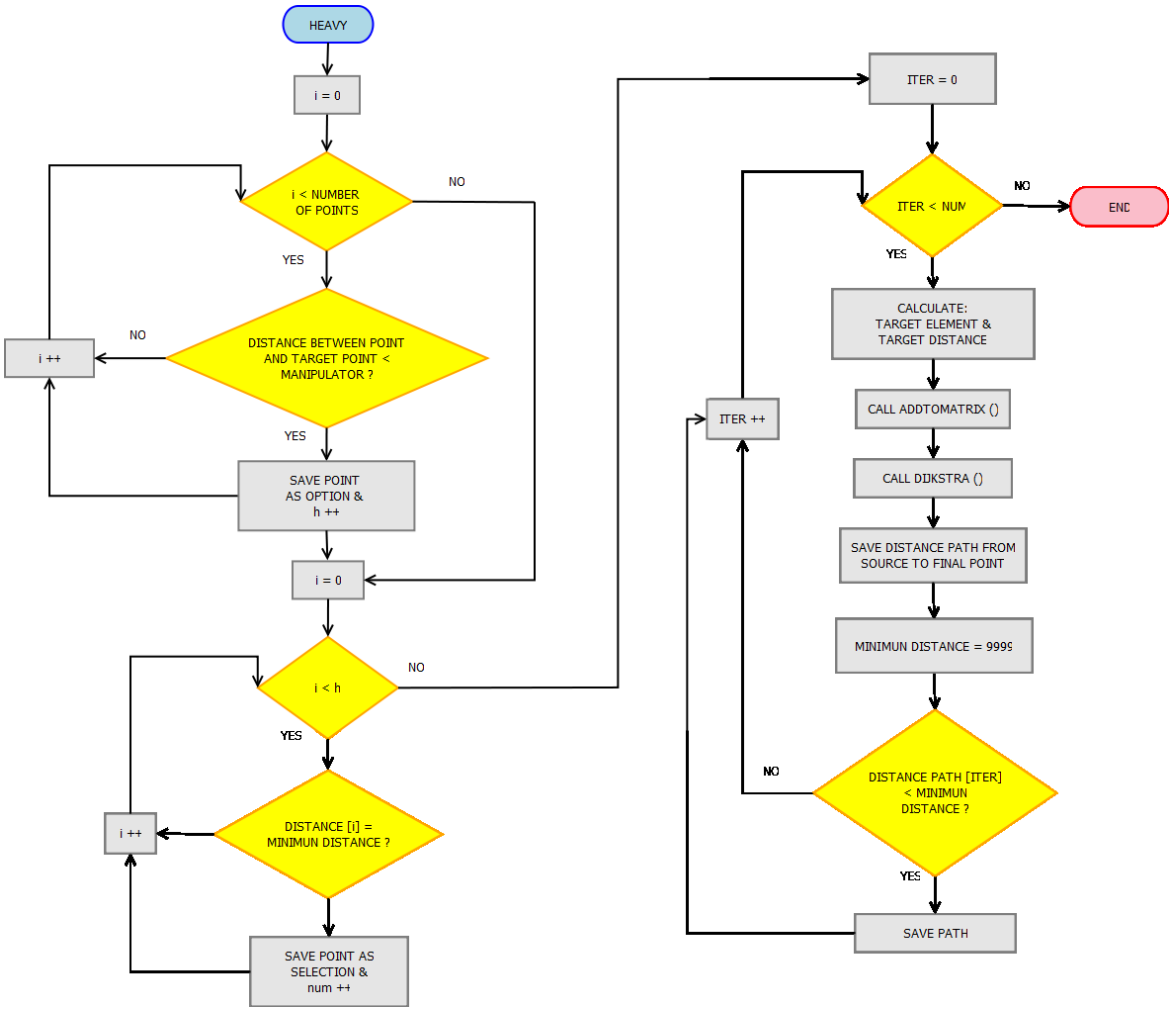


Figure A.7: Heavy Diagram. Algorithm corresponding with a heavy object

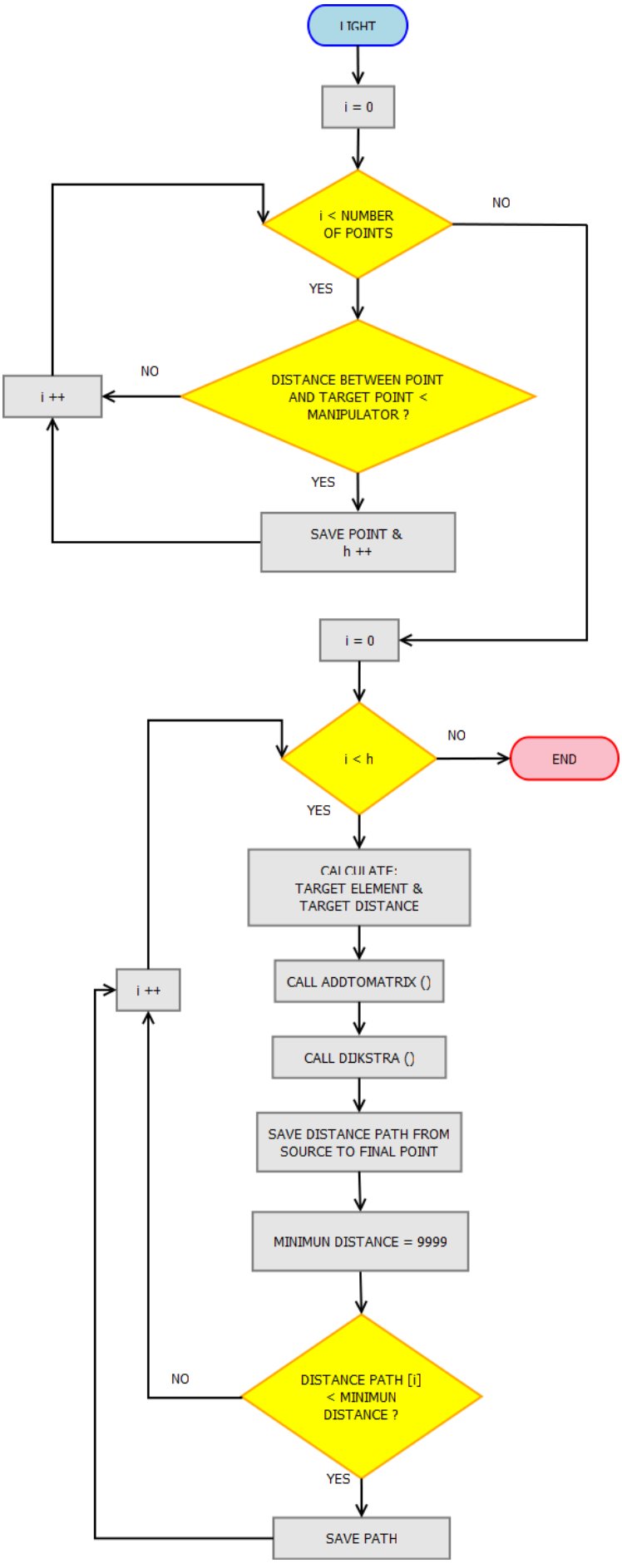


Figure A.8: Light Diagram. Algorithm corresponding with a light object

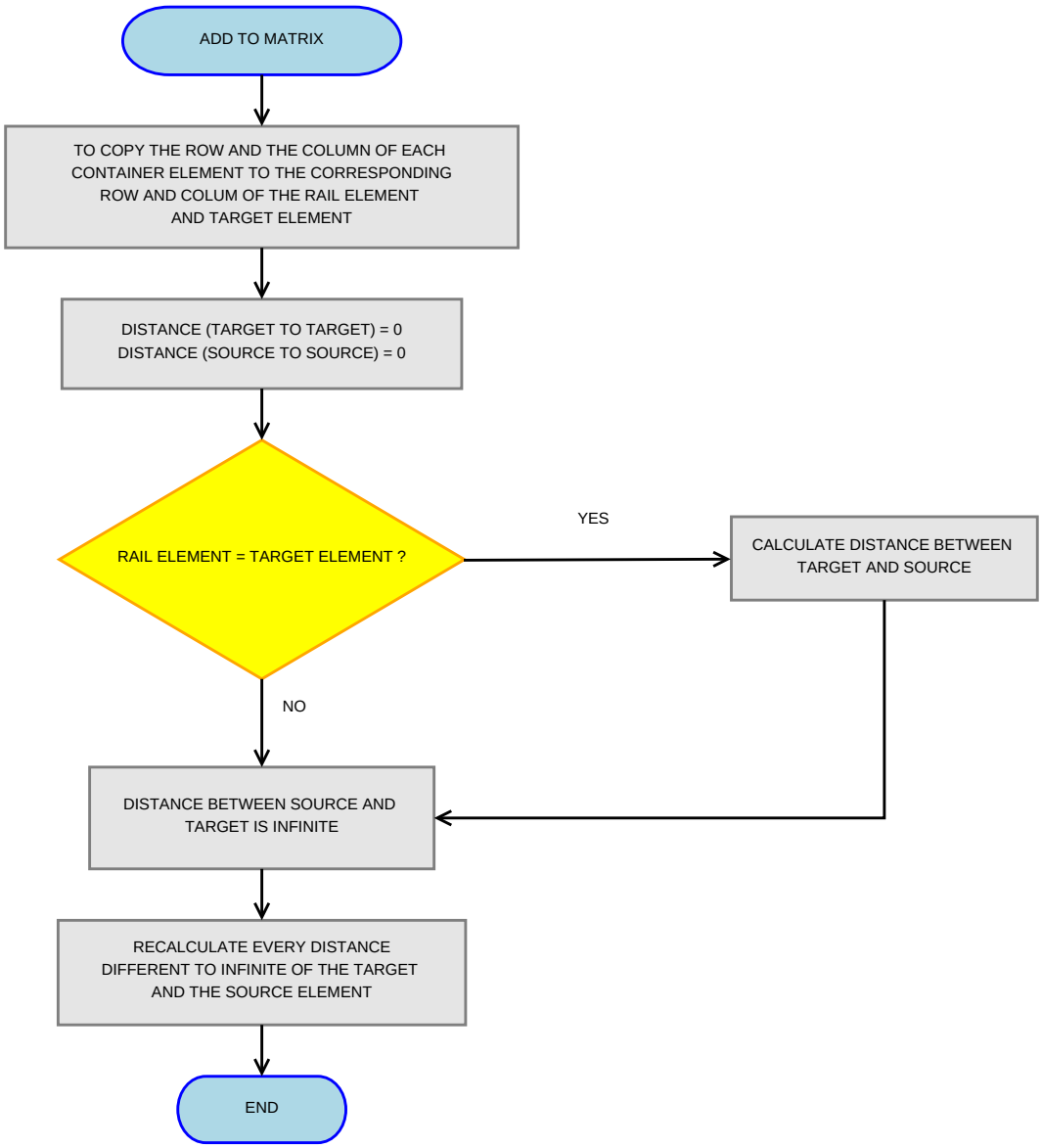


Figure A.9: Add to Matrix Diagram. The Adjacency Matrix is completed with the target and source point

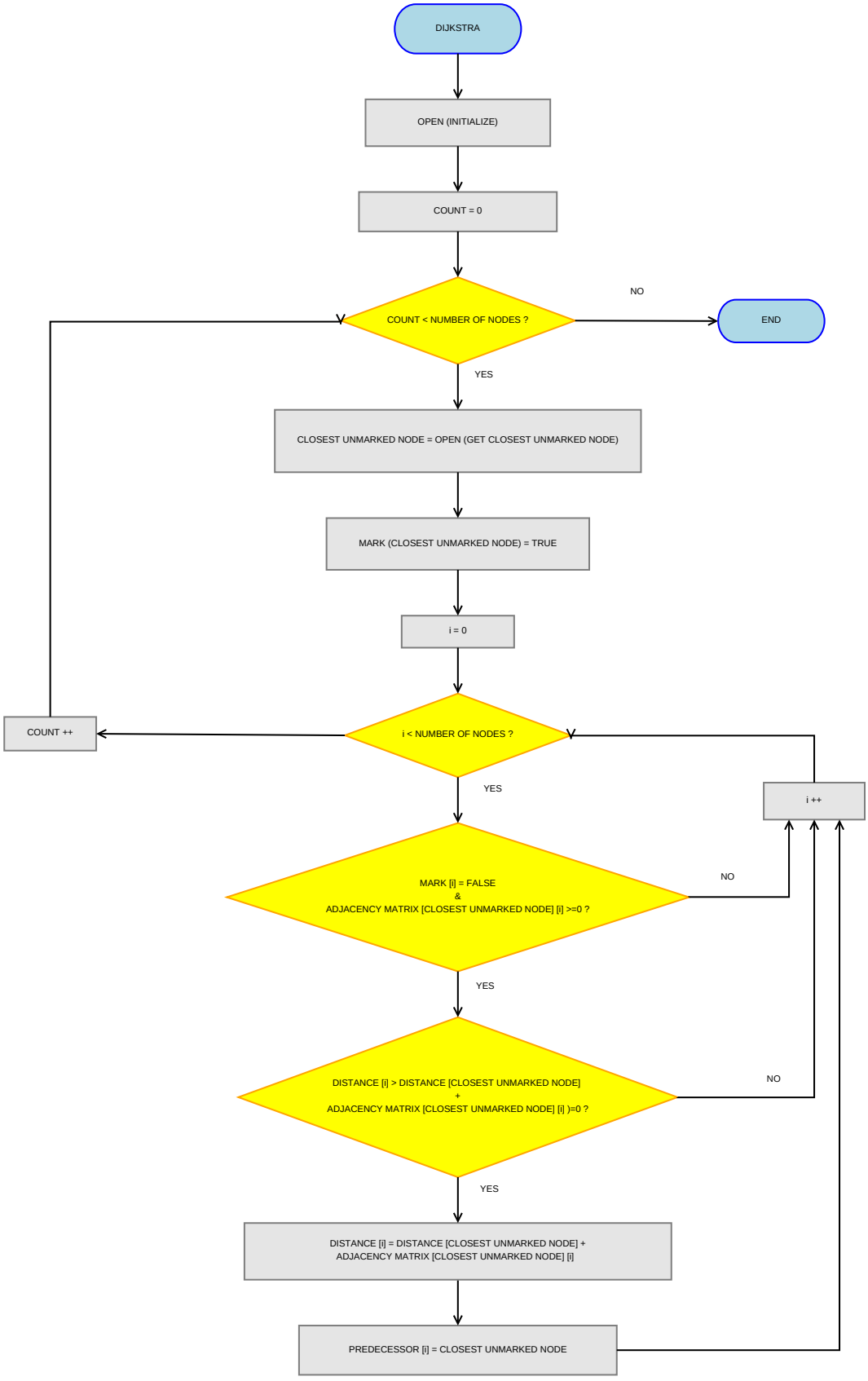


Figure A.10: Dijkstra Diagram. Main program of the Dijkstra Algorithm.

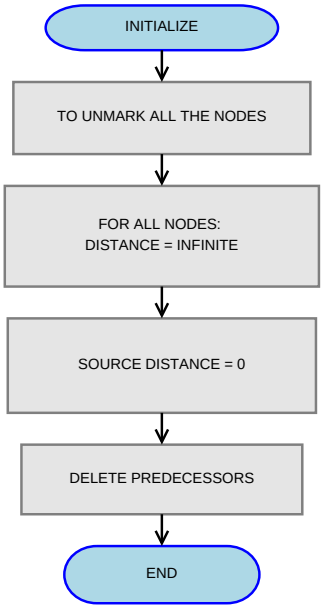


Figure A.11: Initialize Diagram. This code restart the default values for a correct working of the Dijkstra Algorithm

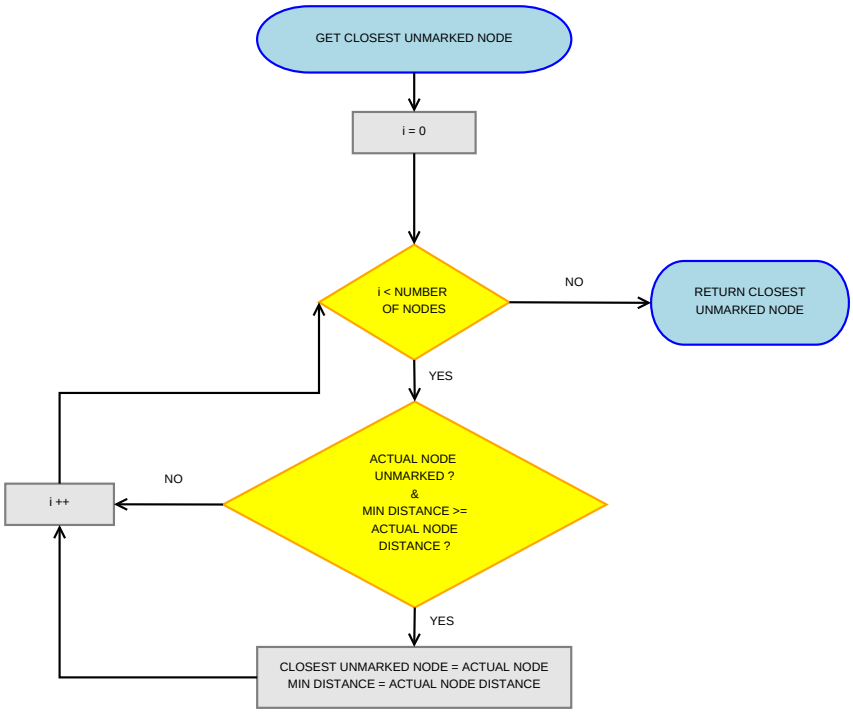


Figure A.12: Get Closest Unmarked Node Diagram. The next unmarked node is checked and marked

Appendix B

Upper Level Software Code

```
#include <stdio.h>
# include <stdlib.h>
# include <iostream>
# include <fstream>
# include "gimi.h"
# include <unistd.h>
# include <cmath>
# include <math.h>
# include "../Wrapper/PositionWrapper.h"

# define INFINITY 9999

const float manipulator = 12;

using namespace std;
using namespace MaCI::Position;

class Graph{
private:
bool mark[15];
int source;
int n;

public:
```

```
float distance[15];
int targetelement;
float targetdistance;
float raildistance;
int railelement;
int numOfVertices;
int rail[100][100];
int route[100];
float adjMatrix[15][15];
int predecessor[15];
float readelements[100][3];
float result[3];
float line[100][4];
int numOfElements;
float X0;
float Y0;
float X1;
float Y1;
float points[2000][3];
bool railfail=0;

void Maci();
void readposition();
void initialize();
int getClosestUnmarkedNode();
void dijkstra();
int open();
int Heavy();
void weightobject();
void addtoMatrix();
void closestPoint(float X1, float X2);
int Gimnet();
int Light();
int RailError();
};
```

int main(){

Main program which decides the logical order of the program.

```
Graph G;
G.open();
while (1) {
G.Gimnet();
G.Maci();
G.weightobject();
}
return -1;
}
```

void Graph::Maci(){

This object initialize the MaCI Position Client. The server is the Field Service Robots Simulator (FSRSim), in concrete the map "fsr2010.xml"

The program is stopped waiting for one event, when it gets the position in x,y coordinates the program continues.

```
PositionWrapper* positionWrapper;
MaCI::Position::CPositionClient* pc;
int timeout;
MaCI::Position::CPositionData posData;
bool posAvailable;
int sequence;

init PositionClient
pc = positionWrapper->getPositionClient();

Attempt to Open Client
MaCI::EMaCIError e;

if ( (e = pc->Open()) != MaCI::KMaCIOK)
{
dPrint(1,"Failed to Open PositionClient instance! (%s)", GetErrorStr(e).c_str());
}
```

```
else
{
timeout = 1000;
sequence = -1;
}

When one position in avainle it gets the position.
posAvailable = pc->GetPositionEvent(posData, &sequence, timeout);
if(posAvailable)
{
TPose2D *pos = (TPose2D*)posData.GetPose2D();
cout « pos->x « " " «pos->y «endl;

The position in save in two global variables, X0 and Y0.
X0=pos->x;
Y0=pos->y;
posAvailable = false;
}
}
```

int Graph::open(){

The object "open" is the one which opens all the text files and save all the information which is going to be necessary. This object is out of the while loop and only is going to be done when the program is starting.

```
numOfVertices=5;
int h=0;
float dummy;
float chain[100];
int k=0;
int i=0;
float reading[5000];
float read[100];


With this file the program reads the adjacency matrix
ifstream f("matrix.txt", ifstream::in);
```

```
if (!f)
{
cout « "fail" « endl;
return -1;
}

f » dummy;

while (!f.eof()){
chain[i]=dummy;
f » dummy;
i++;
}
f.close();


for (int i=0; i<numOfVertices; i++){
for (int j=0; j<numOfVertices; j++){
adjMatrix[i][j]=chain[1+j+10*i];
}
k++;
}


In this file are all the points.
ifstream f2("system.txt", ifstream::in);

if (!f2)
{
cout « "fail" « endl;
return -1;
}
f2 » dummy;

i=0;
while (!f2.eof()){
reading[i]=dummy;
```

```
f2 » dummy;
i++;
}
f2.close();

numOfElements=i/3;
cout«numOfElements«endl;
for (int h=0; h<numOfElements; h++){
for (int j=0; j<=2; j++){
points[h][j]=reading[j+3*h];
}
}
```

In this file are the first point of each element.

```
ifstream f3("elements.txt", ifstream::in);

if (!f3)
{
cout « "fail" « endl;
return -1;
}

f3 » dummy;

i=0;
while (!f3.eof()){
read[i]=dummy;
f3 » dummy;
i++;
}
f3.close();

for (int i=0;i<numOfVertices;i++){
for (int j=0;j<3;j++){
readelements[i][j]=read[j+3*i];
}
}
```

return 0;
}

int Graph::Heavy(){

Heavy is done when the mapping module defines the object that is necessary to carry or get by the Ceilbot as a heavy object. This definition makes that the Ceilbot look for the closest point in the rail-system to the object.

int elm=0;
int h=0;
float option[numOfElements][4];
float distmin;
float distancia;
int num=0;
float selection[numOfElements][4];
float pred[numOfElements][numOfVertices];
int vertice;

cout«"The targetX is "«X1«endl;
cout«"The targetY is "«Y1«endl;

After get the targetX and the targetY is necessary to add this points to the matrix. TargetX and targetY are the target point, but not the target point in the railsystem. The first step is to find that point.

distmin=9999;
numOfElements=400;
cout«"Starting checking the possible points"«endl;

It looks between all the possible points the one which have less distance and save them.
for (int i=0;i<numOfElements;i++){
distancia=sqrt((points[i][0]-X1)*(points[i][0]-X1)+(points[i][1]-Y1)*(points[i][1]-

```
Y1));
```

Is also important to check that the distance to the point in less than the lenght of the manipulator.

```
if (distancia<=distmin && distancia<manipulator){
distmin=distancia;
option[h][0]=points[i][0];
option[h][1]=points[i][1];
option[h][2]=points[i][2];
option[h][3]=distancia;
h++;
}
}
```

Between all the already saved points it looks for the one with the minimun distance.

```
for (int i=0;i<h;i++){
if (option[i][3]==distmin){
cout«"Found point with minimun selected distance"«endl;
selection[num][0]=option[i][0];
selection[num][1]=option[i][1];
selection[num][2]=option[i][2];
selection[num][3]=option[i][3];
num++;
}
}

distmin=9999;
```

Now with this points is necessary to do Dijkstra and check which is the one with less distance.

```
for (int iter=0; iter<num;iter++){
targetelement=selection[iter][2];
targetdistance=sqrt((selection[iter][0]-readelements[targetelement][1])*(selection[iter][0]-
readelements[targetelement][1])+
(selection[iter][1]-readelements[targetelement][2])*(selection[iter][1]-readelements[targeteleme
```

```
cout«"The distance of the target is.."«targetdistance«".in the element."«targetelement«endl;

cout«"The target point is.."«selection[iter][0]«" "«selection[iter][1]«endl;

First is necessary to modify the Adjacency Matrix addtoMatrix();
dijkstra();
cout«"The distance from 0 is.."«distance[0]«endl;
cout«"The distance from 1 is.."«distance[1]«endl;
cout«"The distance from 2 is.."«distance[2]«endl;
cout«"The distance from 3 is.."«distance[3]«endl;
cout«"The distance from 4 is.."«distance[4]«endl;
cout«"The distance from 5 is.."«distance[5]«endl;
cout«"The distance from 6 is.."«distance[6]«endl;


elm=0;

Then we save and check the one with less distance.
if (distance[numOfVertices+1]<distmin){
vertice=numOfVertices+1;
h=0;
while (predecessor[vertice]!=-1){
pred[elm][h]=predecessor[vertice];
vertice=pred[elm][h];
h++;
}
elm++;
}
}

We show by screen the final predecessor. This points would be sended to the
trolley module to know the path.
while (predecessor[elm-1][i]!=-1){
cout«"The predecessor number.."«i«"..is.."«pred[elm-1][i]«endl;
i++;
}
```

return 0; }

void Graph::readposition(){

In this object the position readed by the odometry and sended via MaCI are going to be checked and founded the corresponding point in the railsystem.

```
int elm;
float distancia;
float distmin;

cout«"The position; X "«X0«endl;
cout«"The position; Y "«Y0«endl;

distmin=9999;
numOfElements=400;

We look the closest point in the railsystem. This point should have a distance around zero.
for (int i=0;i<numOfElements;i++){
distancia=sqrt((points[i][0]-X0)*(points[i][0]-X0)+(points[i][1]-Y0)*(points[i][1]-Y0));
if (distancia<=distmin){
distmin=distancia;
elm=points[i][2];

raildistance=sqrt((points[i][0]-readelements[elm][1])*(points[i][0]-readelements[elm][1])+(points[i][1]-readelements[elm][2])*(points[i][0]-readelements[elm][2]));

railelement=elm;
}
}

cout«"The distance of the position is.."«raildistance«"..in the element.."«railelement«endl;
}
```

void Graph::weightobject(){

This object simulates the decision of the mapping module and defines two different behaviours of the robot. In this moment the decision is made by hand.

```
bool weight;
cout«"Select between light object (0) or heavy object (1)";
cin»weight;

if (weight==1){
readposition();
Heavy();
}
else {
readposition();
Light();
}
}
```

void Graph::addtoMatrix(){

The Adjacency matrix is completed adding the information of the target and source point.

The first step is to copy the rows and the columns of the element where each point are.

```
for (int i=0;i<numOfVertices;i++){
adjMatrix[i][numOfVertices]=adjMatrix[i][railelement];
adjMatrix[i][numOfVertices+1]=adjMatrix[i][targetelement];
adjMatrix[numOfVertices][i]=adjMatrix[railelement][i];
adjMatrix[numOfVertices+1][i]=adjMatrix[targetelement][i];
}
```

Distances between one point and itself are zero.

```
for (int i=0;i<=1;i++)
adjMatrix[numOfVertices+i][numOfVertices+i]=0;
```

If the element of both point are the same the distance are the rest between them. Otherwise the distance is infinite.

```
if (railelement==targetelement){
adjMatrix[numOfVertices][numOfVertices+1]=abs(raildistance-targetdistance);
adjMatrix[numOfVertices+1][numOfVertices]=abs(raildistance-targetdistance);
```

```
}
else{
adjMatrix[numOfVertices][numOfVertices+1]=9999;
adjMatrix[numOfVertices+1][numOfVertices]=9999;
}
```

If the distance are different of infinite, is necessary to recalculate it

```
for (int i=numOfVertices;i<numOfVertices+2;i++){
for (int j=0;j<numOfVertices;j++){
if (adjMatrix[i][j]!=9999 && i==numOfVertices){
adjMatrix[i][j]=abs(adjMatrix[i][j]-raildistance);
adjMatrix[j][i]=abs(adjMatrix[i][j]-raildistance);
}
else if (adjMatrix[i][j]!=9999 && i==numOfVertices+1){
adjMatrix[i][j]=abs(adjMatrix[i][j]-targetdistance);
adjMatrix[j][i]=abs(adjMatrix[i][j]-targetdistance);}
}
}
```

In case that one rail element fails, the distances must be infinite.

```
if (railfail==1){
for (int i=0;i<numOfVertices+2;i++){
adjMatrix[i][elementfail]=9999;
adjMatrix[elementfail][i]=9999;
}
}
}
```

void Graph::initialize(){

This code restart the default features of the Dijkstra's Algotithm.

```
for(int i=0;i<numOfVertices+2;i++) {
mark[i] = false;
predecessor[i] = -1;
distance[i] = 9999;
}
```

```
distance[numOfVertices] = 0;
}
```

int Graph::getClosestUnmarkedNode(){

This code looks for the next node which is unmarked.

```
int minDistance = 9999;
int closestUnmarkedNode;
for(int i=0;i<numOfVertices+2;i++) {
if((!mark[i]) && ( minDistance >= distance[i])) {
minDistance = distance[i];
closestUnmarkedNode = i;
}
}
return closestUnmarkedNode;
}
```

void Graph::dijkstra(){

Main Dijkstra's object. Main routes are made in it.

```
initialize();
int closestUnmarkedNode;
int count = 0;
while(count < numOfVertices+2) {
closestUnmarkedNode = getClosestUnmarkedNode();
mark[closestUnmarkedNode] = true;
for(int i=0;i<numOfVertices+2;i++) {
if((!mark[i]) && (adjMatrix[closestUnmarkedNode][i]>0) ) {

if(distance[i] > distance[closestUnmarkedNode]+adjMatrix[closestUnmarkedNode][i])
{

distance[i] = distance[closestUnmarkedNode]+adjMatrix[closestUnmarkedNode][i];
predecessor[i] = closestUnmarkedNode;
}
}
}
```

```
count++;
}
}
```

int Graph::Light(){

Light is done when the mapping module defines the object that is necessary to carry or get by the Ceilbot as a light object. This definition makes that the Ceilbot look for the optimal point in the rail-system to the object.

```
int elm=0;
int h=0;
float option[numOfElements][4];
float distmin;
float distancia;
int num=0;
float selection[numOfElements][4];
float pred[numOfElements][numOfVertices];
int vertice;

cout«"The targetX is "«X1«endl;
cout«"The targetY is "«Y1«endl;

distmin=9999;
cout«"Starting checking the possible points"«endl;
```

It checks all the possible points. For that it saves all the points which have a distance lower that all of the before points.

```
for (int i=0;i<numOfElements;i++){
distancia=sqrt((points[i][0]-X1)*(points[i][0]-X1)+(points[i][1]-Y1)*(points[i][1]-
Y1));
if (distancia<manipulator){
option[h][0]=points[i][0];
option[h][1]=points[i][1];
option[h][2]=points[i][2];
option[h][3]=distancia;
h++;
}
```

```
}

distmin=9999;
The code makes Dijkstra's with all the points and decided the optimal. for (int
iter=0; iter<h;iter++){
targetelement=selection[iter][2];

targetdistance=sqrt((selection[iter][0]-readelements[targetelement][1])*(selection[iter][0]-


readelements[targetelement][1])+(selection[iter][1]-readelements[targetelement][2])*(selection
readelements[targetelement][2]));

cout«"The distance of the target is.."«targetdistance«".in the element."«targetelement«endl;

cout«"The target point is.."«selection[iter][0]«" "«selection[iter][1]«endl;
addtoMatrix();
dijkstra();
cout«"The distance from 0 is.."«distance[0]«endl;
cout«"The distance from 1 is.."«distance[1]«endl;
cout«"The distance from 2 is.."«distance[2]«endl;
cout«"The distance from 3 is.."«distance[3]«endl;
cout«"The distance from 4 is.."«distance[4]«endl;
cout«"The distance from 5 is.."«distance[5]«endl;
cout«"The distance from 6 is.."«distance[6]«endl;


elm=0;
if (distance[numOfVertices+1]<distmin){
vertice=numOfVertices+1;
h=0;
while (predecessor[vertice]!=-1){
pred[elm][h]=predecessor[vertice];
vertice=pred[elm][h];
h++;
}
elm++;
}
```

}

We show by screen the final predecessor. This points would be sended to the trolley module to know the path. while (predecessor[elm-1][i]!=-1){
cout«"The predecessor number.."«i«"..is.."«pred[elm-1][i]«endl;
i++;
}

return 0;
}

int Graph::Gimnet(){
This object received the information of the GUI of the target point. For accomplish this it uses one Gimnet client and it receives one pointer with float information of the X, Y position.

int messageCount = 0;

Create the GIMI-object.
gimi::GIMI gi;

Connect to tcphub on ASRobo
if (!gi.connectToHub("asrobo.hut.fi", 50002, "ClickPositionReceiver")) {
dPrint(1, "Could not connect to tcpHub. Are connection parameters correct?");
return -1;
} else{
dPrint(1, "Successfully connected.");
}

Message-object for receiving. See gimimessage.h for description.
gimi::GIMIMessage message;

Receiver loop that ends after receiving 2 messages.

Calling receive with infinite timeout. It won't return until a message is received (or connection to hub is lost) This receive gets messages with all protocoltypes.

```
int receiveResult = gi.receive(message, -1);

Check if receiving succeeded.
if (receiveResult == GIMI_OK) { dPrint(1, "Received a message!");

Check if message is of type that we can understand.
if (message.getMajorTypeId() == GIMI_PROTOCOL_CEILBOT) {
cout«"Message is a Ceilbot«endl;
printf("It has %d bytes of data. n", message.getDatalength());

Here we typecast the data within the message to the header for easier handling.
float *number = reinterpret_cast< float* >(message.getData());

Now we can access parameters within data via the header.
printf("Ceilbot packet: The number that I have received is.... %f..", *number);
X1=*number;

number++;
printf("Ceilbot packet: The number that I have received is.... %f..", *number);
Y1=*number;
} else {
printf("Message was of unknown type");
}

}else {
Print the received error code and it's description
dPrint(1, "Error %d (%s) occurred, quitting.", receiveResult,
gimi::getGimiErrorString(receiveResult).c_str());
}

Closes logfile in case if it was used. (debug printing)
debugDeinit();
return 0;
}
```

Appendix C

Gimi Wrapper

C.1 GimiWrapper.cpp

```
#include "GimiWrapper.h"

#define GIMI_CLIENT_API 20000

static int verbose = 1;
static volatile bool run = true;
static ownThreadHandle signalHandlerThread;
static bool gimiCreate = false;
static gimi::GIMI g;

GimiWrapper::GimiWrapper()
{ }

gimi::GIMI* GimiWrapper::getGIMI()
{
if(!gimiCreate){
GIMnet parameters

std::string gimnetAP = "asrobo.hut.fi";
int gimnetAPPort = 50002;
```

```
std::string gimnetName = "ClickPositionSender";
std::string gimnetSourceName = "";

debugInit();
debugSetGlobalDebugLvl(1);
debugSetLogFilename("PositionClientExample.log");

int r;
if ( (r = g.connectToHubEx(gimnetAP, gimnetAPPort, gimnetName) ) !=
GIMI_OK)
{
dPrint(1,"Failed to connect to GIMnet AccessPoint '%s:%d' with name '%s':
'%s'",
gimnetAP.c_str(),
gimnetAPPort,
gimnetName.size()?gimnetName.c_str():"<anonymous>",
gimi::getGimiErrorString(r).c_str());
}
else
{
dPrint(2,"Succesfully connected to GIMnet at AccessPoint '%s:%d' with name
'%s': '%s'",
gimnetAP.c_str(),
gimnetAPPort,
gimnetName.size()?gimnetName.c_str():"<anonymous>",
gimi::getGimiErrorString(r).c_str());
}
gimiCreate =true;
}
return &g;
}

GimiWrapper:: GimiWrapper()
{ }
```

C.2 GimiWrapper.h

```
#ifndef __GIMIWRAPPER_H__
#define __GIMIWRAPPER_H__

#include "gimi.h"
#include "PositionClient.hpp"

#include "ImageClient.hpp"
#include "owndebug.h"
#include "ownutils.h"
#include "binbag.h"
#include "ImageClient.hpp"
#include "ImageData.hpp"
#include "ImageContainer.hpp"

class GimiWrapper
{
GimiWrapper();
GimiWrapper();

gimi::GIMI* getGIMI();
};

#endif //__GIMIWRAPPER_H__
```

Appendix D

Position Wrapper

D.1 PositionWrapper.cpp

```
#include "PositionWrapper.h"

using namespace MaCI::Position;
ConnectionParameters: Change here if possible
std::string datasourceMaCISLPosition = "FSRSim_J2B2.MaCI_Position.Motion";
static int verbose = 1;
static volatile bool run = true;
static ownThreadHandle signalHandlerThread;
static bool gimiCreate = false;

static MaCI::Position::CPositionClient *pc;
static bool pcCreate = false;

PositionWrapper::PositionWrapper()
{ }

PositionWrapper::PositionWrapper(std::string maciName)
{datasourceMaCISLPosition = maciName;
}

void PositionWrapper::InitSignalHandler(void)
```

```
{
Now; Set to BLOCK ALL SIGNALS
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);
sigaddset(&mask, SIGTERM);
sigaddset(&mask, SIGPIPE);
sigaddset(&mask, SIGHUP);
pthread_sigmask(SIG_BLOCK, &mask, NULL);

Start signal handler thread
signalHandlerThread = ownThread_Create((void*)SignalHandlerThread, NULL);
}

MaCI::Position::CPositionClient* PositionWrapper::getPositionClient() {
if(!pcCreate)
{ GimiWrapper* gimi = new GimiWrapper();

init the GIMI and Position Client!
pc = new MaCI::Position::CPositionClient(gimi->getGIMI(), 0);

fprintf(stderr, PositionClient - is loading");

MaCI::EMaCIError e;

GIMI is Open(), so we can execute this.
MaCI::MaCICtrl::SMaCISL sl(datasourceMaCISLPosition);

if (sl.IsValidFQMaCISL())
{ pc->SetDefaultTarget(sl, 5000); }
else
{ dPrint(1,"Failed to open datasource! (%s)",
GetErrorStr(e).c_str());
}
}
return pc; }

int PositionWrapper::SignalHandlerThread(void *)
```

```
{ dPrint(8, "Signal handler thread ( %ld) starting...", pthread _self());

while (run) { Now, wait for requested signals to arrive
int sig;
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);
sigaddset(&mask, SIGTERM);
sigaddset(&mask, SIGPIPE);
sigaddset(&mask, SIGHUP);
sigwait(&mask, &sig);

Got signal, sigwait returned!
dPrint(1,"Got Signal '%d' (%s)", sig, strsignal(sig));

switch(sig) {
case SIGINT:
case SIGTERM:
case SIGPIPE:
case SIGHUP:
dPrint(1,"Terminating...");
run = false;

dPrint(1,"Waiting 3 seconds for cleanup...");
ownSleep_ms(2000);
dPrint(1,"Exit.");
ownSleep_ms(1000);
exit(1);
break;

default:
dPrint(1,"Unhandled signal! Ignore!");
break;
}
}
return 0;
}
```

```
PositionWrapper:: PositionWrapper()
{ }
```

D.2 PositionWrapper.h

```
#ifndef__POSITIONWRAPPER_H__
#define __POSITIONWRAPPER_H__

#include "gimi.h"
#include "PositionClient.hpp"
#include "GimiWrapper.h"
#include <string>

class PositionWrapper {
static int SignalHandlerThread(void *);
static void InitSignalHandler(void);


public:
PositionWrapper();
PositionWrapper(std::string);
PositionWrapper();


MaCI::Position::CPositionClient* getPositionClient(); };

#endif // __POSITIONWRAPPER_H__
```